JForth Professional

for the Amiga

User Manual
and
Reference Guide

Version 3.0
January, 1992
Published as freeware in 1997

http://www.softsynth.com/jforth

Technical Support is not Available

Mike Haas

Phil Burk

Brian Donovan

Jim King

with Martin Kees

COPYRIGHT NOTICE AND WARNING

This software package and manual are copyrighted 1986, 1987, 1988,1989 and 1991 by Delta Research or the original author if so specified in the source code file.

This JForth package is released as freeware. Permission is hereby given for any third party to reproduce, distribute and modify the JForth software code or any derivative works thereof without any compensation or license. The JForth software code is provided on an "as is" basis without any warranty of any kind, including, without limitation, the implied warranties of merchantability and fitness for a particular purpose and their equivalents under the laws of any jurisdiction, as well as the provision of support of any kind.

How to Use This Manual

This Manual is organized into four main sections. The first is a tutorial for people who are just learning Forth. If you already know Forth, you may want to just skim the main tutorial until you get to the parts that are unique to JForth. The second section describes the features of JForth as a general purpose programming language. The third section describes the parts of JForth that are specific to the Commodore Amiga. The fourth is an extensive glossary of the words defined in JForth and is intended as a reference.

I strongly recommend at least skimming the whole manual. There are a lot of features of JForth that are easy to miss. There is no point in recreating tools that already exist. I also recommend reading the Table of Contents carefully. I recommend also looking at the description of what files are on the disk to make sure you have not missed anything.

If you are the type of person who learns best by example, you may want to skip to the description of the demos in Chapter 3. You can print the source code for the demos which you will find in the JD: directory on the JTools disk.

Besides this manual, you will also need a good Amiga internals manual to make use of the Amiga libraries. The Intuition manual and the ROM Kernal manuals are particularly important. These are available from Commodore, and most computer book stores.

Be sure to also look at the README files on the disk, and look in the JDOC: directory for any late breaking news. Check out the Bibliography too for good Forth text books.

Instant Gratification

We recommend that you read Chapter 1 first, then do the tutorials. If you simply cannot wait then try the following steps. If you are confused by anything, slow down and go back to the tutorial.

Step 1: Make backups of all disks.

Step 2: From the WorkBench, open the "Extras" disk then open the "Com" drawer.

Step 3: Double Click on the JForth Icon.

Step 4: Wait for JForth to load then, in JForth, enter the following paying close attention to spaces. (Note: The word ." has no spaces between the dot and the quote.)

```
:   HI   ."   Hello World!"  ;
HI
```

Congratulations. You have just written, compiled, linked and tested your first JForth program. JForth routines start with a colon and end with a semicolon.

Step 5: Insert the “JForth” disk and enter in JForth:

```
DOS EXECUTE JFORTH:ASSIGNS
```

Step 6: Now load the Graphics system by entering:

```
INCLUDE  JU:AMIGA_GRAPH
```

Step 7: Now open a test window and draw a line:

```
GR.OPENTEST
20 10 GR.MOVE
123  45  GR.DRAW
```

Step 8: Now close the window and get to work on the tutorials. Enter:

```
GR.CLOSECURW
BYE
```

Acknowledgements

We would like to thank the many people who have contributed to the development of JForth. Special thanks to Martin Kees for his significant contributions to JForth 3.0, to Jack Woehr for his enthusiastic support of Forth in general, and JForth in particular, and for his valuable feedback; to Larry Polansky for his extensive support and early testing; to Marlin Schwanke for testing JForth and for hosting a JForth topic on his BBS at (619)477-2368; to Nick Didkovsky for his help with the Debugger, ARP and his fanatacism; Bill Kelly and Kirk Baker for their contributions of public domain code and extensive testing; to Dave Sirag for his Floating Point implementation; to Bill Maginnis for his many helpful suggestions; to George Kozlowski for his version of U/ and testing; to Lynn Newton for help with the manual, and to our Beta testers - Robert Marsanyi, Curtis Stanton, Saul Lande, Mark Hellman, Chris Greene, Henry Lowengard, Jay Baldock, Robert Dickow, Rene LeBlanc, Kaspar Osterbye, Jay Christensen, David Brown, Peter Yadlowski, Roy Brunges, Jack Johnson and others who have helped make JForth more usable for everybody. Thanks also to the master hackers from Amiga who gave us this great machine.

[Also thanks to Martin Randall for helping us with the release of JForth as Freeware.]

We would also like to thank the neglected friends of us Delta Researchers for being so patient while we buried ourselves in the machine! Sorry if we've left anybody out...it's *not* from lack of appreciation!

JForth Reference Manual
Table of Contents

5 Forth Reference

1) Installation and Startup

Make Backup Copies....1
Installing JForth on a Hard Disk....2
Running JForth from Floppies using the Shell....2
Running JForth from Floppies using the Workbench....3
Tips for Running With Only 512K....3
    Reboot....3
    Do NOT Load the Workbench, use the Shell....3
    Reduce the Size of JForth....4
Running JForth with only 1 Disk Drive....4

2) Introduction to JForth

Major Functional Systems....1

Grand Tour of JForth....3
JForth Compared to other Forths....6

3) JForth Disk Organization

Directory Nicknames....1
CL: - Extras:Clone - Clone Recompiler....1
COM: - Extras:Com - Executable Command Images....1
FD: - JForth:fd.files....2
JA: - Extras:Appls - Applications....2
JARP: - JTools:JARP - ARP Interface....2
JD: - Extras:Demos....2
JDEV: - JTools:DevTools - Development Tools....3
JF: - Extras:Sysgen - System Generation....4
JFLT: - Extras:Floats - Floating Point....5
JI: - JForth:Include....5
JIFF: - Extras:IFF - Interchange File Format....5
JO: - Extras:ODE - Object Development Environment....5
JPICS: - JTools:JPics - Pictures for tutorials....6
JRX: - JTools:JARexx - Arexx Interface....6
JTX: - JTools:Textra_Dir - Text Editor....6
JU: - JForth:Util - Utilities....6
General Forth Utilities....6
Amiga Specific Utilities....7

4) Beginning Forth Tutorial

Forth Syntax....1
The Stack....1
Arithmetic....4
Defining a New Word....5
More Arithmetic....5
Arithmetic Overflow....6

Character Input and Output....6
Answers to Problems....8

5) Intermediate Forth Tutorial

Editing Programs in Files....1
Sample Program....1
INCLUDE the Program....2
Variables....3
Constants....4
Logical Operators....5
Flow of Control....6
Loops....7
Text I/O....9
Changing Numeric Base....10
Answers to Problems....11

6) Advanced Forth Tutorial

String Handling....1
Saving Forth....2
Programming Aids....3
The Forth Interpreter and Dictionary....3
Return Stack....5
Extending the Compiler....6

7) Clone - The JForth Target Compiler

How To Use Clone....1
Technical Information About Clone....2
Clone Glossary....2
Customizing the CLONEd Image....3
Clone Configuration File....5
Word Redefinitions under Clone....6
How To Be Clone Compatible....6
Differences Between Original and Cloned Code....8

8) File I/O

File I/O Tutorial....1
    Creating a Text File....1
    Reading a Text File....2
    Using Binary Data Files....2
File I/O Reference....4
    Opening Files....4
    Reading and Writing to files....5
    Closing Files....5
    Building AmigaDOS Filenames....6

Sequential Virtual File Utilities....7
DOLINES - Easy Text File Processing....8

9) Floating Point Arithmetic

Floating Point Tutorial....1
Simple Arithmetic and Output....1
Transcendental Functions....2
Precision Independent Style....2
Cloning Floating Point Code....2
Floating Point Glossary....3
Floating Point Control....3
Arithmetic Operators....3
Result Flags....3
Transcendental Functions....4
Logical Operators....5
Stack Operators....5
Number Storage....6
Number Conversion Operators....6
Display Operators....6
Display Operators & Variables....6
Number Interpreters....7

10) Object-Oriented Development Environment (ODE)

Philosophy....1
Existing Classes in ODE....1
Hidden Data....1
Generic Messages....2
Tradeoffs....2
Origins of OOP....2
Terminology....2
Turorial 1 - Creating and Using Objects....3
Including ODE....3
Creating an Object, Instantiation....3
Sending Messages....3
Using Arrays....3
Finding an item in an Array....4
Range Checking....4
Freeing Memory in Array Classes....5
}STUFF: and FILL:....5
Tutorial 2 - Early versus Late Binding....5
To Whom It May Concern,....5
Local Variables and Late Binding....6
Tutorial 3 - Using OB.ELMNTS....6
Predefined Classes....8
OBJECT....8
OB.INT - subclass of OBJECT....8
OB.BARRAY - subclass of OBJECT....9

Example of Using Arrays....10
OB.ARRAY....10
Example of Execution Array....11
OB.ELMNTS....11
OB.LIST....14
OB.OBJLIST....14
Dynamic Instantiation using OB.OBJLIST....15
Defining New Classes and Methods....16
Class Definition Glossary....16
Instance Variables....17
Using SELF in Method Definitions....17
Using SUPER and SUPER-DOOPER in Method Definitions....18
Special Methods: INIT:....19
Example Class Definition....19
Example of Creating a Class with Instance Objects....20
Advanced Topics....21
ODE Functions....21
Getting Information About Classes....21
Dynamically Allocated Objects....22
Examining Instance Variables....22
Error Reporting....23
Inheritance....23
Memory Placement for Amiga....23
Cloning ODE Programs using JForth....23
Explanation of ODE Structures Diagram....24

11) Miscellaneous Forth Tools

Memory Allocation....1
Use Memory like a Stack....2
Deferred Words....3
Using DEFER to "vector" code....3
Deferred System Words....3
Potential Problems with Defer....4
Tools for FORGET....5
Local Variables....6
Logging to Files or the Printer....8
Word Usage Analysis....9
Error Handling....9
The dreaded GOTO....9

12) System Internals

USER Variable Data Area....1
Data Stack Area....2
Extensible Dictionary Area....2
Other memory Allocation / Utilization....4
Relocations Table....5
Files and Memory Housekeeping....5

JForth Compiler....5
How to Generate a New JForth System....6

Development Tools

13) Debugging

Tools Overview....1
Debugging Hints....2
Source Level Debugger Tutorial....3
Compiling with DEBUG{....3
Examining Code....3
Stopping with a Breakpoint....4
Stopping with Control-D....5
Debugging a Large Program....5
Debugging a Cloned Program....5
IMMEDIATE Words....6
Source Level Debugger Glossary....6
Debugger One Key Commands....7

14) 68000 Assembly

JForth and 68000 Assembly Language....1
JForth Register Utilization....1
JForth 68000 Forth Style Assembler (RPN)....2
Motorola-Style (Forward-Parsing) Assembler....6
Compiling the Forward Assembler....6
The Forward Assembler Label Field....7
The Forward Assembler Opcode Field....7
The Forward Assembler Operand Field....8
Example of Accessing Structure Members....8
Example of Referencing Variables from an Interrupt....9
The Forward Assembler as a MODULE....9
DISM - JForth Disassembler....9
Compiling the Disassembler....9
Disassembler Output....10
"Automatic" Disassembly Features....10
Disassembling within the JForth Image....11
Disassembling outside of the JForth Image....11
The Disassembler as a MODULE....11

15) Forth BLOCK Environment

AmigaDOS Incompatibilities....1
JForth supplied SCREEN utilities....1
Line Editor Operation and Glossary....2
SCRED ... the JForth SCReen EDitor....3
BLOCK2TEXT....4

16) Precompiled Modules

Modules and SAVE-FORTH....1
Technical Notes on Modules....1
Using the Assembler and Disassembler Modules....2
Using other Modules....2
Files in INCLUDES Module....3
Creating a Custom Module....3

17) Miscellaneous Development Tools

Command Line History....1
    Using the Cursor Keys....1
Vocabularies....3
SHOWHUNKS - for Analyzing Amiga Binary Files....5
JForth Optimizing Compiler Extension....6
PROFILE - Performance Analyser....7

Amiga Interface

18) Amiga Libraries and Structures

Amiga Libraries - Tutorial....1
    Passing Addresses to Library Routines....2
    Getting Values from Library Routines....2
Accessing the Amiga Libraries - Reference....3
    Opening Libraries....3
    Closing Libraries....4
Calling Amiga Libraries....4
    Library Open Verification....5
    CALL modifiers....5
    CALL shortcuts....5
    Adding Libraries....6
Amiga 'C' Structure Interface....7
    Structures in the Amiga....7
    Loading Structure Definitions from ".j" Files....7
    Loading Structure Definitions from Precompiled Modules....7
    Using Structures....8
    Making an Array of Structures....8

Referencing Substructures....8
Accessing Array Members in Structures....9
Examining Structures with DST....9
Defining Your Own Structures....9
Structure Glossary....10
Structure Accessing Words....10
Structure Defining Words....11
Member UNIONs....12
Addressing Considerations - Important!!!....13
H2J - Convert "xx.h" to "xx.j"....14

19) Graphics Toolkit

Graphics Tutorial....1
Generic Graphics Glossary....3
Control Routines....3
Output Primitives....4
Output Attributes....5
Graphics Input....6
Event Driven Programming....6
Routines in JU:AMIGA_EVENTS - EV.xxxx....6

20) EZMenu System

Tutorial....1
EZMenu Structure....1
AMENU Program....1
EZMenu Glossary....4
EZMenu Default Settings....6
Low Level Menu Support....6

21) IFF Support

Description of Files in JIFF:....1
Tutorial 1 - Displaying Pictures....1
Tutorial 2 - The Picture System....2
Drawing a Portion of a Picture....2
Special Effects - Wipes and Fades....3
Moving a Brush, Restoring the Background....4
Cleaning Up....5
Picture System Reference....5
Error Handling....5
Double Buffering....5
Using your Own Display Screen....6
Clipping with Pictures....6
Picture Glossary....6
JIFF:PICTURE....6
JIFF:PIC_EFFECTS....9
JIFF:PIC_FLIP 9

IFF File Support....9
How JForth Handles IFF files....10
Tutorial 3 - Vectored Parsing of IFF Files....10
Printing Chunk Headers....10
Parsing ILBM FORMs....11
IFF Support Glossary....11
JIFF:ILBM_PARSER....11
JIFF:ILBM_MAKER....12
JIFF:SHOW_IFF....13
Low Level Support....14
JIFF:IFF_SUPPORT....14
JIFF:UNPACKING....15
JIFF:PACKING....15
JIFF:PACKING_OLD....16
Incompatibilities with JForth V2.0....16

22) Anims and Animbrushes

ANIM Formats....1
Compiling the ANIM Toolbox....2
Tutorial 1 - Displaying an ANIM File....2
Tutorial 2 - ANIM Control and Disk Based ANIMS....3
Tutorial 3 - ANIMBRUSHES....4
Animation Tips....5
ANIM Support Glossary....6
ANIM IFF tools....6
ANIMATION Words....7
ANIMBRUSH Words....8
CONVERSION Words....9
LOW LEVEL support words....10

23) ARexx Support

What is ARexx?....1
Description of Files....1
Low Level ARexx Support....2
ARexx Toolbox Tutorial....2
ARexx Toolbox Glossary....5
Integrating Textra and JForth....7
ARexx Variables Interface....8
Variable Names....8
The RVI Glossary....8
RVI Code Examples and Test Program....9

RexxView....10

24) Miscellaneous Amiga Support

What is supported and why?....1
File JU:GRAPH_SUPPORT....1
Font Support....3
File JU:GADGET_SUPPORT....3
File JU:POLYGON for Area Fill....4
File JU:SCREEN_SUPPORT....4
Exec Library Support....5
Other Exec words - CreatePort() AbortIO() etc....6
Amiga Linked List Tools....6
ANSI Text and Cursor Control....7
Amiga DOS 2.0 Support....8
Identifying the Workbench Version....8
JU:ASL_SUPPORT....8
ASL Forth Utilities....9
Tag Lists....9

Glossary

Key to Glossary - Stack Diagrams

Appendices

A) Delta Research Biographies....1

B) Resources....2

On-Line....2
Organizations....2
Publications....3

C) ASCII Control Characters....4

D) Sample Applications....5

CR2LF.f - Carriage Return to Line Feed....5
DIAL.f - Quick Dialer using a Modem....5
Docu.f - Automatic "Documentation" Generator....6
DumpBrush.f - Dump a brush as JForth source code....6
DumpIFF.f - Dump IFF file for analysis....6
H2J.f - Convert a 'C' style ".h" file to a ".j" file....6
Print.f - Print a File....6
Rude.f - Print Rude Message using an Alert....6
SortMerge.f - Merge Presorted Files....7
SayNumber.f - Recite a single-precision number in decimal....7
Terminal.f - Very Dumb Terminal Program....7
Update.f - Copy newer files from one directory to another....7
WordCount.f - Count Words Lines and Chars....7

E) Style Guidelines....8

Naming Conventions....8
Transportability Techniques....10
Multistandards....12

F) Words by Function....13

G) Incompatibilities....18

Between V2.0 and V3.0....18
Error Handling in IFF and Picture words....18
Local Variables....18
Between V1.2 and V2.0....18
>BODY and BODY>....18
CONSOLE....19
Totally New Floating Point....19
Bugs Fixed....19
Signed AND Unsigned Structure Members....19

H) Products Written in JForth.....20
B.A.D. from MV Micros.....20
My Diary from MV Micros.....20
HMSL, the Hierarchical Music Specification Language.....20
Copyist Companion by Nick Didkovsky.....20
Doctor Nerve with Nick Didkovsky.....21
IntuiEZ by Curtis Stanton.....21
XL by Martin Kees.....21
JGoodies_1 from various.....21

Chapter 1
Installation and Startup

[Note: These instructions were for the commercial version of JForth. Please refer to the instructions that came with the Freeware archive for the latest information.]

Prerequisites

For the optimal use of JForth we recommend having at least 1 Megabyte of memory and either two floppy drives, or one floppy and a hard drive. It is possible to use some parts of JForth with only 512K of RAM or one disk drive, but limitations are quickly encountered. We will describe later in this chapter how to do that.

We also recommend that you be familiar with the general use of the Amiga and the AmigaDOS operating system. Specifically, you should be familiar with performing the following tasks:

1) Making backup copies of floppy disks,

2) Opening a CLI or Shell window,

3) Using the CLI commands ASSIGN, COPY, DIR, EXECUTE, RENAME and RUN.

If you are not familiar with these, please review the Amiga User Manual and other documentation because an explanation of these is beyond the scope of this manual. You should be comfortable with the operation of the Amiga before attempting any programming. Once you are familiar with Amiga operation, the use of JForth should be quite straight forward.

We recommend that you work from a CLI window or a Shell window. You can also work from the Workbench but this is a less powerful environment for programming.

To open a Shell window in Amiga DOS, open the Workbench or your hard disk icon and double click on the "Shell" icon.

Make Backup Copies

JForth Professional comes on three disks. The "Extras" disk and the "JTools" disk are specifically labeled as such; the remaining disk is simply called "JForth".

The first thing is to make working copies of your disks. You should then keep the original copies as backups. JForth is not copy protected so you can use DISKCOPY from the CLI. If you use the Workbench to make your copies, you will need to relabel the new disks because the Workbench prepends "Copy of" to the disk label. Remove that so that the copies have the same names as the originals. It is probably a good idea to write protect your original disks before starting to make the backups to avoid accidental erasure.

It is also very important to frequently make backup copies of the programs you write. Making backups is important even if you are the kind of person who never makes mistakes. If the power goes out, or some other error occurs, while you are writing to a disk, then you will probably lose at least one file. It

is usually the file you are working on. As long as you make frequent backups of your work, you can minimize your loss.

The steps you take now depend on the configuration of your Amiga system.

If you have a **Hard Disk**, proceed with the section on Hard Disk Installation.

If you only have **One Floppy Disk Drive** and NO hard disk, skip to the section on running with 1 drive, (or buy another disk drive).

If you have two floppy disk drives and at least 1 Meg of RAM, skip to the section on running JForth from Floppies.

Installing JForth on a Hard Disk

We have provided a script file to help users install JForth on a hard drive. It will create a directory called WORK:JFORTH_DIR on your hard disk. It will then copy the three JForth disks into separate directories in the new directory. If you would like to specify a different name for this directory, you can pass a directory name as a parameter to the script.

Note: If you have a previous version of JForth installed, you will need to UN-ASSIGN the JFORTH:, and EXTRAS: logical volumes. Otherwise the system will confuse the JFORTH: on your hard disk with the JFORTH: floppy disk. The best way to UN-ASSIGN these is to remove the ASSIGNs in your startup sequence in the S: directory, then reboot the Amiga. You do not need to remove the old version itself but you may want to rename the directory it is in to avoid naming conflicts with the new version.

To install JForth on a hard disk, put the JTOOLS: disk in a floppy drive then enter in the shell:

```
CD RAM:
COPY JTOOLS:INSTALL_JFORTH RAM:
```

You now have a choice to make. To put JForth in the directory WORK:JFORTH_DIR, enter:

```
EXECUTE INSTALL_JFORTH
```

or, to use a different, directory, enter:

```
EXECUTE INSTALL_JFORTH other_directory_name
```

You will then be asked to insert the three JForth disks, one at a time. It will take several minutes for the files to copy.

You should now modify your startup sequence based on the instructions printed when the script finishes. These modifications will assign logical names to the JForth directories so that JForth will know where to look for its files.

Finally reboot the computer so that the ASSIGNs will be executed. To run JForth you can now enter:

```
RUN COM:JFORTH
```

If you would like to perform a quick test of JForth, wait for JForth to start, then enter in JForth:

```
INCLUDE JD:DEMO_BOXES
```

After compilation is complete, enter:

```
BOXES
```

You should now see a window with random boxes being drawn. Click in the closebox (upper-left) to stop the demo.

To **quit** from JForth, enter:

```
BYE
```

Running JForth from Floppies using the Shell.

Put your working copy of the "JForth" disk in the external drive and enter:

```
EXECUTE JForth:ASSIGNS
```

This sets up logical volumes in the Amiga that will help it keep track of where files are located. Put your JTools disk in the external drive and enter:

```
EXECUTE JTools:ASSIGNS
```

Now put your working copy of the "Extras" disk in the external drive and enter:

```
EXECUTE Extras:ASSIGNS
```

Now, let's run JForth itself. Enter:

```
RUN COM:JFORTH
```

This will load JForth. You will eventually see a new window pop up that has the Delta Research copyright. If you hit a carriage return, you will see an OK message appear. This is your sign that JForth is working.

You can now move on to the tutorials. When you are eventually ready to leave JForth, enter:

```
BYE
```

This will close the JForth window and terminate execution of JForth.

Running JForth from Floppies using the Workbench

Insert your "JForth" disk in any drive. Double click on the disk icon to open it. Now double click on the "**Assignments**" icon. This will run the ASSIGNS command file.

Repeat the above procedure for the "JTools" and "Extras" disks.

Now open the COM drawer on the Extras disk and double click on the JForth icon.

This will load JForth. You will eventually see a new window pop up that has the Delta Research copyright. If you hit a carriage return, you will see an OK message appear. This is your sign that JForth is working.

You can now move on to the tutorials. When you are eventually ready to leave JForth, enter:

```
BYE
```

This will close the JForth window and terminate execution of JForth.

Tips for Running With Only 512K

We recommend having at least 1 Megabyte of memory. The operation of the Amiga is greatly improved by the addition of extra memory. You can use recoverable RAM disks and add extra disk buffers to speed up disk access. With enough memory and AmigaDOS 1.3 you can even reboot from RAM.

If you have an Amiga with 512K of RAM you will be able to run JForth but will not have much memory to spare. To give yourself room for JForth AND a text editor you may have to take some steps to reduce your memory consumption. Here is a list of some things you can do to give yourself more room. You probably do not need to do all of these things. Just pick what you want to try.

Reboot

If you have been running your Amiga awhile, your memory can become fragmented. This is not as bad as it sounds. It just means that the free memory in your system is arranged in many small pieces instead of a few large pieces. JForth needs a large piece to fit in. By rebooting, you can reorganize the free memory on your system into larger pieces. I recommend getting a copy of the program AVAIL which will tell you how much memory you have and how big the largest piece is.

Rebooting will also clear out any fonts or libraries that other programs opened and left laying around in memory.

Before rebooting first make sure that all your applications have stopped because anything in RAM will be lost. To reboot, hold down the <CTRL> key and the two <AMIGA> keys simultaneously.

Do NOT Load the Workbench, use the Shell

Another way to save memory, about 13K, is to give up using icons. You can prevent Workbench from loading by modifying your startup sequence to open a Shell instead of the Workbench.

Edit the file S:STARTUP-SEQUENCE. Change the following text at the end of the file:

```
LOADWB
```

to be:

```
; LOADWB
NEWSHELL
```

Adding a semicolon will comment out a line.

Reduce the Size of JForth

The original JForth comes with 64K of free dictionary. If you don't need this much you can reduce the size of your JForth image by changing #K and using SAVE-FORTH. Use a BACKUP copy of JForth for this exercise. Enter in the CLI:

```
EXECUTE EXTRAS:ASSIGNS
RUN COM:JFORTH
```

Now enter in JForth:

```
#K @ .  \ print how much
#K @ 20 - #K !  \ subtract 20K
SAVE-FORTH COM:JForth
BYE
```

Now enter in the CLI:

```
RUN COM:JFORTH
```

The new image will use about 20K less memory. Some large programs may not load in this smaller dictionary space. Please see SAVE-FORTH in the glossary for more information about this technique.

Running JForth with only 1 Disk Drive

The Amiga operates best with 2 floppy disk drives or 1 floppy and a hard disk. Cost is a limiting factor for many people, however, so we have made provisions for running from 1 drive.

The most important thing to do when running from 1 drive is to limit the number of disk swaps you have to make. The Amiga will ask for the Workbench disk when it needs to load a command from the C: directory. You can avoid many of the requests for the Workbench disk by placing the necessary commands in RAM. The Amiga will be able to load these commands from RAM and not need the Workbench disk as often. (Please note that since these commands will take up RAM space so you might have problems if you only have 512K of memory.)

The commands that should be in RAM:, or made RESIDENT, are CD, DIR, RUN, EXECUTE, ASSIGN, LIST and COPY.

Chapter 2
Introduction to JForth

JForth is a programming language that allows you to interact with your Amiga. When you are programming in JForth, you are "inside" the language. You can access any data structure, test any routine, or use any development tool, right from the keyboard. This direct communication with the computer can improve your productivity, giving you additional time to improve the quality of your software products.

JForth is based on the 1983 standard Forth language. Forth was first conceived by Charles Moore when he wanted a new language for controlling telescopes. He developed a language based on a dictionary of words. This dictionary can be extended by writing new words based on the old ones. Since Mr. Moore's original version, Forth has been translated to almost every type of computer from the biggest mainframes to the smallest microcomputers.

Since a minimal Forth can be implemented in just a few kilobytes of memory, it is often used in very small embedded computer systems for process control and robotics. Forth, however, is equally appropriate for larger, more advanced, computers like the Amiga. Forth is a very flexible language and can be adapted to larger computers without losing the flavor of the original language.

JForth is an implementation of Forth designed specifically for the Commodore Amiga. JForth uses a 32-bit stack and compiles directly to 68000 machine code. This makes JForth faster than most Forths. JForth also provides an extensive set of tools for accessing the special features of the Amiga. You can call any Amiga Library routine by name and reference any Amiga structure using constructs similar to 'C'. JForth also has some special toolboxes that support simple graphics, Intuition menus, IFF files, and other Amiga features. These toolboxes can be used directly to simplify your Amiga programming. Source code for all these toolboxes is provided so you can customize them if needed or study them as examples of Amiga programming. JForth also provides over a dozen small sample programs for those, like me, who learn best by example.

JForth also allows you to do things that are unique in the Forth experience, the most dramatic being CLONE. This exeptional utility allows you to create a totally independant, standalone version of your program of minimal size that is entirely royalty free!

Major Functional Systems

For a full list of the tools that JForth offers, you should read the next chapter about the various files on the disks. The systems mentioned here are ones that we feel are major ones not to be missed.

Amiga Library Calls

JForth provides a system for easily calling any Amiga Library routine by name. It will figure out what 68000 registers the parameters go into and build the appropriate code. Thus you can simply pass parameters on the stack in the order described in the Amiga documentation. A number of toolboxes have been written for supporting specific parts of the Amiga Library including EZMENUS, Graphics,

the Serial Device, ANSI codes, and much more. Chapter 18 covers the details of Amiga library access.

Amiga Structure Support

JForth provides the equivalent to all of the ".h" include files from 'C'. These ".j" files define all of the necessary structures and constants for passing information to the Amiga Library routines. Structures can be dumped interactively with all members shown by name and value for debugging by using DST. Chapter 18 further discusses this helpful feature.

ARexx Support

ARexx is a language that helps various applications communicate with each other. A spread sheet program, for example, could communicate with a data base program. These tools help you write ARexx compatible programs. Chapter 23 further discusses AREXX.

Assembler

JForth support two 68000 assemblers, one with Reverse Polish Notation (RPN) and one with Motorola like syntax. The RPN assembler can be used to create macros but the Motorola assembler is easier to read. We also offer a Disassembler. All are covered in depth in Chapter 14.

Block Support and SCRED

For those who prefer the old fashioned BLOCK or SCREEN environment, we provide LIST, LOAD and the standard line editor. We also provide a WYSIWYG SCREEN editor called SCRED. (We recommend the use of normal text files with JForth). Chapter 15 discusses the BLOCK environment.

Clone

Clone can be used to generate small, royalty free, executable images of your JForth programs. Clone will take a compiled JForth program, extract out all of the code and data needed to run it, and reassemble a smaller version of it. All of the JForth development tools, the name fields and link fields and any other unused words are left behind. The final image size is comparable to images created using a 'C' compiler and linker. Images can be saved with a symbol table for use with WACK or other debuggers (if needed). The JForth Source Level Debugger can also be used with Cloned programs. Most programs will Clone without modification if they follow a few simple rules regarding run time initialization of variables or arrays containing Forth addresses. See Chapter 7 for further information.

Debugger

JForth provides a source level debugger that allows you to single step through your code. At every step you can see what is on the stack. You can also examine the return stack, dump memory, set breakpoints, control execution, or even enter Forth commands. Chapter 13 fully documents the source level debugger.

Floating Point

JForth support both the single precision Fast Floating Point and the IEEE double precision. These words conform to the Forth Vendors Group Standard. Chapter 9 describes the Floating Point implementation.

IFF Support

JForth provides general purpose tools for reading and writing IFF files. It also provides a toolbox specifically for ILBM graphics files. This allows you to use pictures, brushes, anims and animbrushes from a paint program, or other source, in your programs. JForth also provides tools for animation and presentation such as, blit, wipe, fade in, fade out, etc. Powerful graphics and animation programs can be created using these tools. Chapter 21 will further guide you in this area.

Local Variables

Local variables can simplify the definition of complex words by eliminating much of the stack manipulation. Local variables are fast self fetching variables that allow reentrant recursive code to be written. Using regular VARIABLEs can make a word non reentrant. Local variables are specifically covered in Chapter 11.

Modules

JForth's precompiled modules provide a method for rapidly accessing code that is used during complation. This includes the Assemblers and Disassembler, and the Amiga include files. These modules are dynamically linked into the dictionary when needed. This way, they do not take up space in your normal dictionary, yet are instantly available. See Chapter 16 for more information about modules.

MultiStandard

This system allows you to switch easily between JForth and the major standards - FIG , '79 , and '83. This is handy for compiling code written using other Forths. Multistandard is discussed in Chapter 17.

ODE

ODE is an Object Oriented Development Environment similar in concept to SmallTalk. It allows you to define classes of intelligent data structures, then create as many copies of these data structures as needed. This technique can simplify programming immensely by making it easier to write reusable code. Chapter 10 is devoted to ODE.

Profiler

This optimization tool monitors the operation of a program to find out where it is spending its time. Most programs spend most of their time in a small portion of the code By identifying this code, one can focus efforts at optimization where they are most needed. The profiler is documented in Chapter 17.

Textra

Textra is a powerful, yet easy to use, multi-window text editor designed for programmers. It uses the mouse to select text and allows Cut, Copy and Paste, operations between its windows. It also incorporates ARexx hooks that allows you to use provided macros (.textra files) for text processing (or write your own). Textra can communicate with JForth such that you can compile directly from the editor. If an error is encountered while compiling, Textra will highlight the error so the you can fix it quickly. The documentation for Textra may be found in the JTX:DOCS directory. To access the '.textra' REXX scripts enter:

```
COPY JRX:SCRIPTS/#? REXX:
```

Grand Tour of JForth

The section describes some of the special features of JForth. If you have never used any Forth, you may want to try the tutorials first. Then come back to this chapter because it describes features you should know about.

JForth has so many tools and utilities that it is easy to miss something useful. We recommend studying this manual carefully so that you can take advantage of all that JForth has to offer. You should also read some of the source code files where you may find some undocumented goodies.

Let's start with a grand tour of the main JForth tools. Bring up JForth and try out these commands as we discuss them. (If you are unclear about how to read stack diagrams, please try the tutorial first.)

WORDS (-- , list the words currently available)

This command lists the words in the dictionary that you can call. Although thousands of words may be listed, you will only need to learn a few. You can stop the output of this list by hitting a key. Hit return to continue listing or enter "quit" to stop. You could also enter a Forth command while WORDS is paused. This pause feature is provided by ?PAUSE which is described in the glossary. VLIST is a synonym for WORDS.

WORDS-LIKE (<word-fragment> -- , list all with fragment)

Use this if you want to see all of the words that have some word fragment in their name. Enter:

```
WORDS-LIKE  EMIT   ( all words with EMIT )
WORDS-LIKE  +      ( all words with '+' )
```

As a shortcut, if you hit <Function-Key-6> it will insert WORDS-LIKE into your input stream. This will save you from having to type it in. Hit the <HELP> key to find out about other function key assignments. Now hit the <Up-Arrow> key repeatedly to scroll through your previous commands just like in the Amiga DOS shell. You can look in the section on Command Line History for more information about this.

FILE? (<name> -- , show file and source code)

If you are curious about how a word is defined you can use FILE? to see what file it is from. It will then ask you if you want to see everywhere it is referenced or defined in that file. This will work for every word in JForth except those from the Assembly Language Kernal which is proprietary. (Hey! We gotta have some secrets!)

```
FILE? ANEW
```

EACH.FILE? (<name> -- , show every file and source code)

If a word has been redefined, this command will find every occurrence of it in the dictionary and show it to you.

```
EACH.FILE?  AUTO.INIT
EACH.FILE?  [FORGET]
```

DEF (<name> -- , disassemble word)

Here is a handy word for those of you who are familiar with 68000 assembly language. DEF will disassemble any Forth word. (See the chapter on the 68000 assemblers for more info.)

```
DEF 1+
DEF CMOVE
```

The first time you enter DEF it will load a precompiled MODULE containing the disassembler. This gives you quick access to DEF without taking up room in the dictionary or having to compile it yourself.

DOS (<command-line> -- , pass command to Amiga DOS)

You can execute almost any Amiga DOS command line except those that require input, and CD. For CD, just enter CD without using the DOS keyword (CD has been provided for you in the JForth dictionary). Some common DOS commands have been predefined. (See the file JF:DOSCOMMANDS for more information.)

Execute arbitrary DOS command

```
DOS TYPE S:STARTUP-SEQUENCE OPT H
```

For CD, use CD directly.

```
CD RAM:
```

Use predefined command.

```
DIR DF0:
```

Pass command in string.

```
" RENAME RAM:MOO RAM:GOO"  $DOS
```

MEASURE (<Forth-line> -- , time Forth commands)

This is handy when you are trying to speed up your code. This command will tell you how long things take. It will take the commands that follow, execute them and tell you how long it took.

```
MEASURE DIR
```

BENCH (<Forth-line> -- , time commands with overhead)

MEASURE and BENCH are only accurate to about 1/50th of a second. If something takes less than 1/2 a second you should call it N times in a loop then divide by N to find out the actual time. If you want to take the loop overhead into account, use BENCH and BENCH.WITH.

```
: TDO  ( N -- )
    0 DO LOOP
;
: TSWAP ( a b count -- a b | b a )
    0 DO SWAP LOOP
;
1,000,000 BENCH.WITH TDO
11 22 33 1,000,000 BENCH TSWAP
```

INCLUDE (<filename> -- , compile source code from a file)

You can specify a complete pathname or use the current directory. INCLUDE can be nested so you can call INCLUDE in the file you are currently INCLUDEing. (See INCLUDE? in the glossary.) INCLUDE is a very important word so we have assigned it to <Function-Key-1>.

```
INCLUDE JU:RANDOM
6 CHOOSE .  ( random number, 0 <= R < 5 )
6 CHOOSE .

TYPEFILE JU:LOGTO
```

```
MEASURE INCLUDE JU:LOGTO
```

JForth uses a HASHED dictionary to speed up compilation. This is a special way of looking words up directly instead of scanning the entire dictionary. If we turn off hashing we can see how hashing speeds up the compile process...

```
HASH.OFF
MEASURE INCLUDE JU:LOGTO
```

Notice the difference in compile speed. Hashing does not, however, come without a price. JForth has to build a table of pointers for hashing to work. This rehashing is required whenever you FORGET code or do a GETMODULEs. If we recompile LOGTO with hashing on we will have to rehash.

```
HASH.ON
MEASURE INCLUDE JU:LOGTO
```

This is still faster than compiling with hashing off. Hashing is most effective with large files when the dictionary has lots of words already defined. Some people have reported an 8-9 times speed up. Hashing is least effective when you recompile small files but they are pretty fast anyway. Hashing will speed up compilation in almost every instance so we usually leave it on.

MAP (-- , display system status)

This word tells you all about the current system, how many files are open, how much room is left in the dictionary, etc. This is assigned to a function key. Hit <HELP> to find out which one.

SAVE-FORTH (<filename> -- , save current JForth system)

If you compile some code and want to save the dictionary in its current state, use SAVE-FORTH to write it to a file. You can run this image later, just like you did COM:JForth. If you try this you should write it to a work disk (NEVER write to the original JForth disks). If you want the new image to have more dictionary space available, increase the value of the variable #K before you SAVE-FORTH...

```
#K @ .   ( see how much is there )
50 #K +! ( increase it by whatever you think you need )
SAVE-FORTH  WORK:MY4TH
```

Be mindful that when you SAVE-FORTH, any opened files or allocated memory will not be preserved in the saved-image, so your programs should re-establish such resources in an INIT-type word. Unless your program meticulously cleans-up after itself, you should SAVE-FORTH after immediately compiling and before running it so that you save a pristine image. See the glossary for more information.

JForth Compared to other Forths

This section describes some of the ways in which JForth differs from other Forths.

32 Bit Stack

Forth uses what is called a "data stack" for passing values between words. It is a place to put numbers, addresses or other values. Values can be added to the stack and removed just like on a stack of plates. Most Forth implementations use a 16 bit stack. This means that the numeric range in a standard Forth is -32,768 to 32,767. JForth and a few other Forths, use a 32 bit stack which gives you a numeric range

of -2,147,483,648 to 2,147,483,647. Since JForth addresses are also 32 bit, you can address a much larger memory space than a 16 bit Forth.

Jump Subroutine Threading

Another major difference between JForth and most other Forths is that JForth is a true compiler. In most Forths,when you "compile" a word, you actually end up creating a table of tokens or pointers to the other words that are called. If your word calls '+' it will put a pointer to the plus function in your word. When you execute your word, a program called the inner interpreter examines these pointers and then executes the appropriate functions That is the traditional method.

In JForth we use what is called Jump Subroutine, or JSR, Threading. That is where the 'J' in JForth comes from. In JSR threading, actual 68000 machine code is compiled into your word. If you call a word, JForth will compile a 68000 JSR instruction to make that call. This eliminates the need for an inner interpreter. If the subroutine is small enough, JForth will copy the called routine inline into your routine thus avoiding the small overhead of the JSR instruction. Since the 68000 executes code directly, JForth will run 2-3 times faster than a traditional Forth on the same machine.

Relative Addressing

In JForth, addresses are expressed as offsets relative to the base of the JForth image in memory. This is important because AmigaDOS will load JForth at different places in memory at different times. Despite this, your programs can use the same relative addresses at different times. This can simplify Forth programming because variables, CFAs, dictionary links, and other addresses keep the same relative addresses even thought their absolute addresses may change.

Amiga Libraries uses absolute addresses. You will, therefore, have to convert relative addresses to absolute before passing them to the Amiga. This is a simple operation. On the whole, we feel using relative addressing is an advantage over absolute addressing.

Here are the words used to convert addresses:

```
>REL  ( absolute-address -- relative-address , convert )

>ABS  ( relative-address -- absolute-address , convert )
```

(Also remember that the addresses on the return stack will be absolute as well. We have provided words for accessing "inline" data based on reading the return stack. Please see the appendix on Transportability.)

Text File Input

JForth will compile programs from "normal" ASCII text files created using standard Amiga text editors. Thus you can use you favorite editor with JForth instead of the TEXTRA editor provided. (Traditional Forths use a system of 1024 byte source code blocks. We provide support for this system for those who want it.)

NOT versus 0=

In the Forth '83 standard, NOT performs a 1's complement operation. Since NOT is usually used for negating a logical value, this can sometimes give surprising results. In Forth, any non zero value will be considered true by IF and other conditional words. The traditional NOT only works with a true value of -1. For that reason, JForth defines NOT as 0= which negates any true value.

```
In Forth '83
        -1 NOT ( is false )
         1 NOT ( is true!!, = -2 )
         1 0=  ( is false )

In JForth
        -1 NOT ( is false )
         1 NOT ( is false )
         1 0=  ( is false )
```

The JForth word COMP will perform a 1's complement operation like the '83 NOT. If you want the traditional NOT you can redefine it or use the Multistandard package which gives you strict conformance with '83, FIG or '79 standards.

Floored Division

JForth division is not floored. This means that the results of a division are rounded up if they are negative. In floored division the result is rounded toward negative infinity.

```
In JForth:   -13 2 / .  ( gives -6 )
Floored:     -13 2 / .  ( gives -7 )
```

Chapter 3
JForth Disk Organization

JForth is released on three disks named *JForth*, *Extras* and *JTools* [The Freeware version is released in a single archive.]

Each disk contains several directories, each containing a specific set of related files. Only four of the directories do not contain JForth program source files; COM: holds only executable binary images, FD: stores information used by the JForth compiler to interface with the Amiga ROM Kernel, and MOD: holds precompiled code.

Directory Nicknames

JForth utilizes features provided by the AmigaDOS ASSIGN command to create *nicknames* for each of the normal directory pathnames. By executeing (another AmigaDOS command) the *assigns* file that is on the root of each disks, you can *teach* your system these nicknames. This need only be done once per session, when you first insert the disks in your system. If you install JForth on your hard disk as described in chapter one, then these assigns will be done automatically.

There are a number of benefits to using assigned names:

1) They short and require less typing then the full path names.

2) You can always use the same name regardless of the actual pathname of the directory. You can always reference JU:LOCALS with the same name regardless of whether JU: is JForth:Util or WORK:Programming/JForth_Dir/JForth/Util.

Please read about ASSIGN in the Amiga DOS manual if you are not familiar with this very powerful command.

If you enter an assigned name and Amiga DOS asks you to enter a disk with the logical name, then your assigns are not set properly. Check the installation procedure to make sure you have executed the assigns properly. If you are not using a hard disk you can simply enter:

```
EXECUTE JFORTH:ASSIGNS
EXECUTE EXTRAS:ASSIGNS
EXECUTE JTOOLS:ASSIGNS
```

Once this has been done, JForth will be able to locate all required source files.

The following is a description of each JForth directory in alphabetical order. Be sure to read the description of JU: since it contains the most generally useful utilities.

CL: - Extras:Clone - Clone Recompiler

CLONE is compiled by loading CL:TOPFILE. Clone is used to generate stand-alone royalty free applications.

COM: - Extras:Com - Executable Command Images

JKERNAL - This image is the basic Forth engine and primitives. Once executed from AmigaDOS, files from the 'JF:' (Extras:SysGen) directory are INCLUDEd to generate a new, fully-operational, JForth image. Since a completely compiled development image is supplied (see 'JForth', below), this need only be done if the files in the 'JF:' directory have been altered by the programmer.

*** JFORTH** - This is the main JForth executable image. This file is comprised of the kernel image (described above) and utilities compiled from the JF:' directory, notably the disassembler, timing words, a pattern-matching 'VLIST', Hashing, History and others. See the chapter on "System Internals" for more information on recompiling COM:JForth.

FD: - JForth:fd.files

This directory contains all the information that the JForth compiler needs to create the proper *glue* routines to interface to any and all Amiga library calls. As supplied, the files in this directory describe the 2.0 version of the AmigaDOS operating system.

JA: - Extras:Appls - Applications

We have provided several complete applications written in JForth. They can be used either directly from JForth or they can be cloned and used as CLI commands. Some of them demonstrate how to parse the command line, how to handle input errors and printing "help" information. The file names all end with ".f" to distinguish the Forth source file from the cloned image file.

These are described in more detail in an appendix. Please see the chapter on Clone for instructions on how to clone these applications.

JARP: - JTools:JARP - ARP Interface

ARP stands for Amiga DOS Replacement. It is a popular library of calls

ARP.J - Include file for interfacing to ARP library.

ARPFileRequest.f - Simple call to ARP's file requester.

JD: - Extras:Demos

This directory contains the JForth source code for all of the supplied demo programs. You can include each demo individually or include most of them at once by INCLUDEing JD:LOAD_DEMOS. At the end of the compile, instructions on how to execute the demos is printed.

These demos illustrate how to use various aspects of the JForth programming environment and the Amiga. You may find these useful as a starting point for your own programs. Feel free to modify these demos and include parts of them in your commercial products.

BENCH_PRIMITIVES - benchmark programs for primitive operations. Use this to compare JForth with other Forths.

DEMO_BOXES - This program is similar to the BOXES demo that comes with the Amiga. It shows you how to open a window and draw random rectangles. It also illustrates using the XOR mode of drawing, and changing colors.

DEMO_CLICK - Shows how to detect double clicks.

DEMO_CIRCSQ - Graphical pattern generator using algebraic functions.

DEMO_DBUF - Shows how to create a double buffered video display.

DEMO_DOTS - Simple random pixels.

DEMO_FONTS - Shows how to open and use Amiga fonts.

DEMO_GADGET - Shows how to use most of the gadget types available in a simple "do-nothing" program.

DEMO_GSORT - An animated demonstration of using the Batcher's sort. It generates random length rows and sorts them visually.

DEMO_HAM - This illustrates opening a CUSTOM SCREEN in HAM mode. A BACKDROP window is then opened within it and some random 'art' is generated.

DEMO_INTERRUPT - Shows how to install an interrupt handler. Uses the Vertical Blanking Interrupt to generate a 60 hertz clock. See also **HIGH_INTERRUPT** which shows a way to call high level Forth code from an interrupt (not recommended).

DEMO_LINES - This is similar to the LINES demo from Commodore, except it stays within the window. It generates sweeping line patterns in different colors.

DEMO_MENUS - The use of the EZMENU system is shown here. A set of menus that demonstrate mutual exclusion, checking, setting of command keys, etc. is built and used.

DEMO_MSG - Open a message port and send a message.

DEMO_PAINT - This is a simple paint program that demonstrates the use of graphics input events. A call to EV.GETCLASS is made. The colors are cycled automatically. Detects and responds to double clicks.

DEMO_POLYGON - Use the AREADRAW routines to draw polygons.

DEMO_RAIN - This doesn't teach anything that BOXES doesn't already show but it looks interesting so here it is.

DEMO_RAWKEY - Shows how to get RAWKEY events from a window and convert them.

DEMO_RGB - Cycle colors by changing the color map of a screen.

DEMO_SCROLL - The demo shows you how to make a direct call to an Amiga graphics library routine, ScrollRaster. It also uses line drawing.

DEMO_SPEAK - This small demo illustrates the use of the Amiga Translator library and Narrator device to create a synthetic dialog. You should also examine the file JU:SPEAK for an example of opening a device and calling DoIO and other Exec calls.

DEMO__SPRITE - This demonstrates the creation and use of a hardware sprite. It bounces around on the screen leaving a textile like pattern. If you let it go for a long time it will slowly change the color scheme. You can modify the width and height to get different patterns.

DEMO_STRIP - No this is not pornography! Displays a simple running strip chart.

Fibonacci - Generates the Fibonacci numeric sequence.

HIGH_INTERRUPT - How to call high level Forth code from a 68000 interrupt. Be careful.

LOAD_DEMOS - This contains a menu based program that integrates all of the other demos. This could be used by retailers for customer demos.

SIEVE - The legendary Sieve of Eratosthenes. Generates prime numbers which is sometimes useful for something besides benchmarking compilers.

JDEV: - JTools:DevTools - Development Tools

BLOCK - Support for old style Forth BLOCK or SCREEN environment.

BLOCK2TEXT - Convert an old style Forth BLOCK based file to a normal text file.

*** DEBUGGER** - Source level debugger. This allows you to single step through your code while examining the stack, etc.

EDITOR - Old style FIG Forth line editor. As in Leo Brodie's starting Forth. We recommend using the more modern text editors or SCRED.

PROFILE - Analyses code performance so you know what to optimize.

SCRED - Screen based block editor, WYSIWYG. Almost enough to make you want to use BLOCKs.

UNUSED - Mark words as used when compiling a file. Handy for stripping old extraneous code out of a system.

WORDCACHE - Caches recently entered words for word completion using a function key.

JF: - Extras:Sysgen - System Generation

The JF: files are necessary for any JForth image...they are loaded on top of the com:JKernal image to produce the com:JForth image, which may be further customized as the programmer sees fit for his particular programming needs.

These files allow the programmer to modify the JForth system at a relatively low level; he may alter and regenerate all but the bottom 23K of code which was written in assembler.

Here is a partial listing of those files.

$TABLE - String Table, string arrays used by disassembler.

.IF - Conditional compilation.

@BITS -

AJF_DICT - low level support for C_STRUCT and ODE.

ANSI - Support for the ANSI terminal escape sequences for fancy screen displays. Change colors, underline, italics, move cursor, etc.

ASM - Reverse Polish 68000 assembler. In a precompiled module.

AUTO - Contains an AUTO.INIT that provides a way of passing a command to JForth from the CLI.

BUILDSYS - lowest level Forth code. Everything below this was written in assembler. Contains IF ELSE THEN and other important stuff.

CALLS - support calls to Amiga libraries.

CASE - CASE OF ENDOF ENDCASE conditionals.

C_STRUCT - Support for the Amiga 'C' structures.

CONDITION - Exotic conditional construct similar to CASE.

DISM - Disassembler. Shows you the 68000 code of a word.

DOSCOMMANDS - Support easy DOS calls from Forth, eg. DIR , COPY

FORWARD-ASM - Motorola style forward assembler. This uses the more familiar 68000 assembler syntax.

HASHING - Hashes the Forth dictionary for faster compilation.

HISTORY - Provide Command Line History and hot Function Keys.

LOADJFORTH - Loads the com:JForth image from com:Minimum. You may want to customize this and recompile.

MAKEINCLUDES - Compile the includes module. You may want to add more to this.

MEASURE - Measure how long it takes to execute something. Great for running benchmarks.

MEMBER - Support for structure members , ..@ and ..@

MODULE - Support for precompiled modules of code like the INCLUDES.

SELECT - Positional case statement. Warning - will not clone! Use CASE or a run-time initialized jump table.

STRING-INTERPRET - Interpret a string containing Forth code.

STRIP-PATHNAME - Strip the path, if any, from a filename.

TRAPS - Trap 68000 exceptions to eliminate some GURU meditation errors.

WORDS-LIKE - Searches dictionary words that contain a substring. Very handy if you can't remember exactly how a word was spelled or for finding related words.

Y-OR-QUIT - Ask the user to answer yes, no or quit.

JFLT: - Extras:Floats - Floating Point

FLOAT.FFP - Single Precision Floating Point Math

FLOAT.DOUBLE - Double Precision Floating Point Math

JI: - JForth:Include

The Include directory contains the JForth-equivalent of the conventional C-language ".h" files. The same file and directory structure has been maintained; the only difference is that the filename suffix is '.j'. The files may be compiled via the normal JForth 'include' command; they are referenced with the nickname 'ji:'. Some of these files have been precompiled into a module called INCLUDES.

JIFF: - Extras:IFF - Interchange File Format

This directory contains the files to support the IFF standard. This is the standard that is used for exchanging pictures, samples or other information between different programs. See chapters 21 and 22. See also JANIM: in this chapter.

DOUBLE_BUFFER - Support for creating double buffered displays which can result in smoother animations.

IFF.J - Structure definitions and Chunk ID definitions, eg. 'FORM', are defined here.

IFF_SUPPORT - Miscellaneous tools for recursively parsing an IFF file.

ILBM_MAKER - Tools for creating an InterLeaved BitMap file.

ILBM_PARSER - Tools for parsing/reading an InterLeaved BitMap file.

LOAD_PIC - Loads Picture system.

PACKING - Routines for packing Bitmaps into BODY chunks and for packing CTABLES into CMAP chunks.

PIC_EFFECTS - Special effects for IFF Pictures like wipes and fades.

PIC_FLIP - reverse and flip a picture.

PICTURES - Picture Animation that allows you to load IFF pictures and perform simple animations, page flipping, transparent blitting, etc.

SHOW_IFF - Example program to display an ILBM graphics picture, plus tools for transparently copying brush bitmaps to a screen.

UNPACKING - The assembly routine for unpacking a Run Length Encoded bitmap, an Uncompressed bitmap, and a CMAP.

TEST_PIC - Simple test/demo of picture system.

JO: - Extras:ODE - Object Development Environment

This directory contains source code for ODE, the Object Oriented Development Environment. Please see the chapter on ODE.

JPICS: - JTools:JPics - Pictures for tutorials

These IFF pictures are used by the test programs and tutorials of the IFF and ANIM Support systems.

JRX: - JTools:JARexx - Arexx Interface

Tools for writing ARexx compatible applications. See chapter on ARexx for a description of the files.

JTX: - JTools:Textra_Dir - Text Editor

Textra is a public domain text editor written in JForth by Mike Haas. It can be integrated with JForth using ARexx. See the docs on disk and the chapter on ARexx interfacing. A subdirectory in JTX: contains scripts which can be copied to your REXX: directory for use with Textra.

JU: - JForth:Util - Utilities

This directory contains a wide variety of tools and utilities. We have divided them into two categories, general purpose Forth utilities and Amiga specific utilities. Files that we think are particularly important are marked with a single asterisk, '*'.

Note: Some of the files that were in JU: in Version 2.0 had to be moved because of disk space constraints. Look in JF:, JDEV:, and JFLT: for these files.

General Forth Utilities

.RSTACK - Dump return stack, like .S .

*** BSEARCH** - Binary search using a deferred word.

*** BSORT** - Batcher sort for sorting anything pretty quick.

CATCH - A way to trap errors and jump back to the caller.

*** CHAR-MACROS** - 'C' like character routines, ISDIGIT , TOUPPER

*** DOLINES** - Provides a simple way of processing the lines of a file using a deferred word.

*** LOCALS** - Local variables. These are handy for reducing the amount of stack manipulation in your program, ie. fewer DUPs , SWAPs and ROTs.

*** LOGTO** - Copy the output of JForth to a file. This can be used for echoing to the printer if you use PRT: as the file.

MORE-ARRAYS - Different types of arrays, 2D arrays, record arrays, etc.

MSEC - Millisecond timer, not very accurate.

MODULEFIND - More module support.

*** MULTISTANDARD** - Provides compatibility with Forth-79, FIG and Forth '83 standards. Look here for common words you expected to find in JForth.

RAM-WORDS - Handy word for using RAM: when editing large files.

*** RANDOM** - 16 bit pseudo-random number generator.

RELOAD - Reload a part of a program that uses multiple files.

SFA_BITS - defines bits in Size Field

SHOWHUNKS - Dump out the hunks in an Amiga binary file using the Disassembler. Great for exploring how things work.

SQRT - Calculate integer square roots fast.

STACKUTILS - Dynamically expanding stacks. A useful data structure.

TURNKEY - Save a JForth image in a royalty free format. These files are big so only use this as a last resort if your program won't Clone.

*** UNRAVEL** - Examine the return stack and tell you who called who. Useful for debugging.

*** VALUE** - Self fetching variable. Like a constant you can change.

Amiga Specific Utilities

*** AMIGA_EVENTS** - Support for getting IDCMP events.

*** AMIGA_GRAPH** - Library of primitive graphics routines to get you started.

*** AMIGA_MENUS** - The EZMenu system which makes it much easier to use text based Intuition pull down menus.

ASL_SUPPORT - Calls to Application Specific Library plus special support for a file and font requester. These only work under Amiga DOS 2.0.

ANSISUPPORT - More general support for ANSI code.

AUTO_REQUEST - Easy way to put up an Amiga Auto Requestor.

CIAB_RSRC.F - Calls into CIAB resource for parallel port interfacing.

COLORDUMP -

CONSOLESUPPORT - Many tools for handling Console devices and RAW windows.

DEVICE-CALLS - The special device calls, BEGINIO() and ABORTIO()

DOS-SUPPORT - Some handy DOS tools, Lock() , Examine() , ExNext().

*** DUMP_STRUCT** - Dump the contents of a structure with the member names.

ERROR_CODES - Defines some sequential error return codes.

EXAMINE - Print information about a file, eg. protection, size, date.

EXEC_LISTS - Linked list macros.

*** EXEC_SUPPORT** - Various EXEC library calls, eg. DoIO() , FindTask() , WaitPort() , etc. Also the 'C' routines that are not in a library like CreatePort() , DeletePort() , CreateExtIO() , etc.

EZALERT - Simple interface to Amiga Alert. Looks like a GURU message.

FDUMP - Dump the contents of a file interactively. Like DUMP but in a file.

FILE-TOOLS - More DOS file tools like FSIZE which gets file size. Read warning in file before using.

GADGET_SUPPORT - Tools for initializing and using Gadgets.

GETCC - Commodore approved way of getting 68000 condition code register contents.

GOTO_ERROR - Provides a jump to an ERROR: label in a colon definition.

*** GRAPH_SUPPORT** - Miscellaneous graphics calls, bitmap tools.

ICON-SUPPORT - Calls for Free Get and PutDiskObject()

POLYGON - Use Area draw routines to draw polygons, GR.TRIANGLE

SCREEN_SUPPORT - NewScreen.Setup , OpenScreen() , Bitmap>Screen, Screen>View, etc.

SERIAL - Support for RS232 serial device.

SET-ICON - Associate an action with an icon.

SPEAK - Simple interface to the narrator and translator libraries.

SPRITES - Support for hardware sprites.

Chapter 4
Beginning Forth Tutorial

Intended Audience

The intent of this tutorial is to provide a series of experiments that will introduce you to the major concepts of Forth. It is only a starting point. Feel free to deviate from the sequences I provide. A free form investigation that is based on your curiosity is probably the best way to learn any language. Forth is especially well adapted to this type of learning. There are a number of excellent text books on Forth that you may want to use along with this tutorial. Look in the bibliography for a list of books. We recommend either *Starting Forth* or *Forth: A Text and Reference*.

In the tutorials, I will print the things you need to type in upper case, and indent them. You can enter them in upper or lower case. At the end of each line, press the RETURN (or ENTER) key; this causes JForth to interpret what you've entered.

Forth Syntax

Forth has one of the simplest syntaxes, or formats, of any computer language. The syntax can be stated as follows, "Forth code is a bunch of words with spaces between them." This is even simpler than English! Each *word* is equivalent to a function or subroutine in a language like 'C'. They are executed in the order they appear in the code. The following statement, for example, could appear in a Forth program:

```
WAKE.UP  EAT.BREAKFAST  WORK  EAT.DINNER  PLAY  SLEEP
```

Notice that WAKE.UP has a dot between the WAKE and UP. This is because it is *one* Forth word with a specific meaning. Forth word names can have any combination of letters, numbers, or punctuation. We will encounter words with names like:

```
." #S SWAP MOVE ! @ . *
```

They are all called *words*. The word **$%%-GL7OP** is a legal Forth name, although not a very good one. It is up to the programmer to name words in a sensible manner.

The Stack

The Forth language is based on the concept of a *stack*. Imagine a stack of blocks with numbers on them. You can add or remove numbers from the top of the stack. You can also rearrange the order of the numbers. Forth uses several stacks. The *Parameter Stack* is the one used for passing data between Forth words so we will concentrate our attention there. The Parameter Stack is also known as the Data Stack. The *Return Stack* is another Forth stack that is primarily for internal system use. In this tutorial, when we refer to the "stack," we will be referring to the Parameter Stack.

The stack is initially empty. To put some numbers on the stack, enter:

```
23 7 9182
```

You will notice that the OK message now has a '3' in front of it. This tells you that there are three numbers on the stack.

Let's now print the number on top of the stack using the Forth word '**.**', which is pronounced " dot ". This is a hard word to write about in a manual because it is a single period.

Enter: **.**

You should see the last number you entered, 9,182 , printed. Forth has a very handy word for showing you what's on the stack. It is **.S** , which is pronounced "dot S". The name was constructed from "dot" for print, and "S" for stack. If you enter:

```
.S
```

you will see your numbers in a list. The number at the far right is the one on top of the stack.

You will notice that the 9182 is not on the stack. The word ' . ' removes the number on top of the stack before printing it. In contrast, ' .S ' leaves the stack untouched.

We have a way of documenting the effect of words on the stack with a *stack diagram*. A stack diagram is contained in parentheses. In Forth, the parentheses indicate a comment. In the examples that follow, you do not need to type in the comments. When you are programming, of course, we encourage the use of comments and stack diagrams to make your code more readable. In this manual, we often indicate stack diagrams in **bold text** like the one that follows. Do not type these in. The stack diagram for a word like ' . ' would be:

. (N -- , print number on top of stack)

The symbols to the left of -- describe the parameters that a word expects to process. In this example, N stands for any integer number. To the right of --, up to the comma, is a description of the stack parameters when the word is finished, in this case there are none because 'dot' "eats" the N that was passed in. (Note that the stack descriptions are not necessary, but they are a great help when learning other peoples programs.)

The text following the comma is an English description of the word. You will note that after the -- , N is gone. You may be concerned about the fact that there were other numbers on the stack, namely 23 and 7 . The stack diagram, however, only describes the portion of the stack that is affected by the word. For a more detailed description of the stack diagrams, there is a special section on them in this manual right before the main glossary section.

Between examples, you will probably want to clear the stack. If you enter **0SP**, pronounced "zero S P", then the stack will be cleared.

Since the stack is central to Forth, it is important to be able to alter the stack easily. Let's look at some more words that manipulate the stack. Enter:

```
0SP .S    \ That's a 'zero' 0, not an 'oh' O.
777 DUP .S
```

You will notice that there are two copies of 777 on the stack. The word **DUP** duplicates the top item on the stack. This is useful when you want to use the number on top of the stack and still have a copy. The stack diagram for DUP would be:

DUP (n -- n n , DUPlicate top of stack)

Another useful word, is **SWAP**. Enter:

```
0SP
23 7 .S
SWAP .S
SWAP .S
```

The stack diagram for SWAP would be:

SWAP (a b -- b a , swap top two items on stack)

Now enter:

```
OVER .S
OVER .S
```

The word **OVER** causes a copy of the second item on the stack to leapfrog over the first. It's stack diagram would be:

OVER (a b -- a b a , copy second item on stack)

Here is another commonly used Forth word:

DROP (a -- , remove item from the stack)

Can you guess what we will see if we enter:

```
0SP 11 22 .S
DROP .S
```

Another handy word for manipulating the stack is **ROT**. Enter:

```
0SP
11 22 33 44 .S
ROT .S
```

The stack diagram for ROT is, therefore:

ROT (a b c -- b c a , ROTate third item to top)

You have now learned the more important stack manipulation words. You will see these in almost every Forth program. I should caution you that if you see too many stack manipulation words being used in your code then you may want to reexamine and perhaps reorganize your code. You will often find that you can avoid excessive stack manipulations by using *local or global VARIABLES* which will be discussed later.

If you want to grab any arbitrary item on the stack, use **PICK** . Try entering:

```
0SP
14 13 12 11 10
3 PICK .    ( prints 13 )
0 PICK .    ( prints 10 )
4 PICK .
```

PICK makes a copy of the Nth item on the stack. The numbering starts with zero, therefore:

```
0 PICK   is equivalent to DUP
1 PICK   is equivalent to OVER
```

```
PICK  ( ... v3 v2 v1 v0 N -- ... v3 v2 v1 v0 vN )
```

(Warning. The Forth-79 and FIG Forth standards differ from the Forth '83 standard in that their PICK numbering starts with one, not zero.)

I have included the stack diagrams for some other useful stack manipulation words. Try experimenting with them by putting numbers on the stack and calling them to get a feel for what they do. Again, the text in parentheses is just a comment and need not be entered.

```
DROP  ( n -- , remove top of stack )

?DUP  ( n -- n n | 0 , duplicate only if non-zero, '|' means OR )

-ROT  ( a b c -- c a b , rotate top to third position )

2SWAP ( a b c d -- c d a b , swap pairs )

2OVER ( a b c d -- a b c d a b , leapfrog pair )

2DUP  ( a b -- a b a b , duplicate pair )

2DROP ( a b -- , remove pair )

NIP   ( a b -- b , remove second item from stack )

TUCK  ( a b -- b a b , copy top item to third position )
```

Problems:

Start each problem by entering:

```
0SP 11 22 33
```

Then use the stack manipulation words you have learned to end up with the following numbers on the stack:

```
1)      11 33 22 22

2)      22 33

3)      22 33 11 11 22

4)      11 33 22 33 11

5)      33 11 22 11 22
```

Answers to the problems can be found at the end of all the tutorials.

Arithmetic

Great joy can be derived from simply moving numbers around on a stack. Eventually, however, you'll want to do something useful with them. This section describes how to perform arithmetic operations in Forth.

The Forth arithmetic operators work on the numbers currently on top of the stack. If you want to add the top two numbers together, use the Forth word **+** , pronounced "plus". Enter:

```
2 3 + .
2 3 + 10 + .
```

This style of expressing arithmetic operations is called *Reverse Polish Notation,* or *RPN*. It will already be familiar to those of you with HP calculators. In the following examples, I have put the algebraic equivalent representation in a comment.

Some other arithmetic operators are **- * /** . Enter:

```
30 5 - .  ( 25=30-5 )
30 5 / .  ( 6=30/5 )
30 5 * .  ( 150=30*5 )
30 5 + 7 / .  \  5=(30+5)/7
```

Some combinations of operations are very common and have been coded in assembly language for speed. For example, **2*** is short for 2 * . You should use these whenever possible to increase the speed of your program. These include:

```
1+  1-  2+  2-  2*  2/
```

Try entering:

```
10 1- .
7 2* 1+ .  ( 15=7*2+1 )
```

One thing that you should be aware of is that when you are doing division with integers using / , the remainder is lost. Enter:

```
15 5 / .
17 5 / .
```

This is true in all languages on all computers. Later we will examine **/MOD** and **MOD** which do give the remainder.

Defining a New Word

It's now time to write a *small program* in Forth. You can do this by defining a new word that is a combination of words we have already learned. Let's define and test a new word that takes the average of two numbers.

We will make use of two new words, **:** ("colon"), and **;** ("semicolon") . These words start and end a typical *Forth definition*. When you type this in, you may want to time how long it takes between hitting the RETURN key and the appearance of the OK prompt. This will be how long it takes Forth to compile and link this small program. Enter:

```
: AVERAGE  ( a b -- avg )  + 2/  ;
```

Congratulations. You have just written a Forth program. If you are used to 'C', you were probably wondering what to get from the kitchen while compiling. One danger of programming in Forth is that, once hooked, you may never take time to eat.

Let's look more closely at what just happened. The colon told Forth to add a new word to its list of words. This list is called the Forth dictionary. The name of the new word will be whatever name

follows the colon. Any Forth words entered after this will be compiled into the new word. This continues until the semicolon is reached which finishes the definition.

Let's test this word by entering:

```
10 20 AVERAGE .  ( should print 15 )
```

Once a word has been defined, it can be used to define more words. Let's write a word that tests our word.. Enter:

```
: TEST ( --)  50 60 AVERAGE . ;
TEST
```

Try combining some of the words you have learned into new Forth definitions of your choice. If you promise not to be overwhelmed, you can get a list of the words that are available for programming by entering:

```
WORDS
```

You can stop this listing by hitting the space bar. To continue, hit the return key. To stop, type:

```
QUIT
```

Don't worry, only a small fraction of these will be used directly in your programs.

More Arithmetic

When you need to know the remainder of a divide operation. /MOD will return the remainder as well as the quotient. the word MOD will only return the remainder. Enter:

```
0SP
53 10 /MOD .S
0SP
7 5 MOD .S
```

Two other handy words are **MIN** and **MAX** . They accept two numbers and return the MINimum or MAXimum value respectively. Try entering the following:

```
56 34 MAX .
56 34 MIN .
-17 0 MIN .
```

Some other useful words are:

```
ABS    ( n -- abs(n) , absolute value of n )

NEGATE ( n -- -n , negate value, faster then -1 * )

ASHIFT ( n c -- n*(2**c) , arithmetic shift of n )

SHIFT  ( n c -- n' , logical shift of n )
```

ASHIFT can be used if you have to multiply quickly by a power of 2 . A negative count is like doing a divide. This is much faster than doing a regular multiply and should be used whenever possible. Try entering:

```
: 256*   8 ASHIFT ;
3 256* .
```

Arithmetic Overflow

If you are having problems with your calculation overflowing the 32-bit precision of the stack, then you can use ***/** . This produces an intermediate result that is 64 bits long. Try the following three methods of doing the same calculation. Only the one using */ will yield the correct answer, 5197799.

```
34867312 99154 * 665134 / .
34867312 665134 / 99154 * .
34867312 99154 665134 */ .
```

Convert Algebraic Expressions to Forth

How do we express complex algebraic expressions in Forth? For example: 20 + (3 * 4)

To convert this to Forth you must order the operations in the order of evaluation. In Forth, therefore, this would look like:

```
3 4 *  20 +
```

Evaluation proceeds from left to right in Forth so there is no ambiguity. Compare the following algebraic expressions and their Forth equivalents: (Do **not** enter these!)

(100+50)/2 ==> 100 50 + 2/

((2*7) + (13*5)) ==> 2 7 * 13 5 * +

If any of these expressions puzzle you, try entering them one word at a time, while viewing the stack with .S .

Problems:

Convert the following algebraic expressions to their equivalent Forth expressions. (Do **not** enter these because they are not Forth code!)

(12 * (20 - 17))

(1 - (4 * (-18) / 6))

(6 * 13) - (4 * 2 * 7)

Use the words you have learned to write these new words:

```
SQUARE   ( N -- N*N , calculate square )
DIFF.SQUARES ( A B -- A*A-B*B , difference of squares )
AVERAGE4  ( A B C D -- [A+B+C+D]/4 )
HMS>SECONDS ( HOURS MINUTES SECONDS -- TOTAL-SECONDS , convert )
```

Character Input and Output

The numbers on top of the stack can represent anything. The top number might be how many blue whales are left on Earth or your weight in kilograms. It can also be an ASCII character. Try entering the following:

```
72 EMIT  105 EMIT
```

You should see the word "Hi" appear before the OK. The 72 is an ASCII 'H' and 105 is an 'i'. EMIT takes the number on the stack and outputs it as a character. If you want to find the ASCII value for any character, you can use the word ASCII . Enter:

```
ASCII W .
ASCII %   DUP .   EMIT
```

```
ASCII A   DUP .
32 + EMIT
```

There is an ASCII chart in the back of this manual for a complete character list.

Notice that the word ASCII is a bit unusual because its input comes not from the stack, but from the following text. In a stack diagram, we represent that by putting the input in angle brackets, <input>. Here is the stack diagram for ASCII.

ASCII (<char> -- char , get ASCII value of a character)

Using EMIT to output character strings would be very tedious. Luckily there is a better way. Enter:

```
: TOFU ." Yummy bean curd!" ;
TOFU
```

The word **."** , pronounced "dot quote", will take everything up to the next quotation mark and print it to the screen. Make sure you leave a space after the first quotation mark. When you want to have text begin on a new line, you can issue a carriage return using the word **CR** . Enter:

```
: SPROUTS ." Miniature vegetables." ;
: MENU
    CR TOFU CR SPROUTS CR
;
MENU
```

You can emit a blank space with **SPACE** . A number of spaces can be output with SPACES . Enter:

```
CR TOFU SPROUTS
CR TOFU SPACE SPROUTS
CR 10 SPACES TOFU CR 20 SPACES SPROUTS
```

For character input, Forth uses the word **KEY** which corresponds to the word EMIT for output. KEY waits for the user to press a key then leaves its value on the stack. Try the following.

```
: TESTKEY ( -- )
    ." Hit a key: " KEY CR
    ." That = " . CR
;
TESTKEY
```

```
EMIT   ( char -- , output character )
KEY    ( -- char , input character )
SPACE  ( -- , output a space )
SPACES ( n -- , output n spaces )
ASCII  ( <char> -- char , convert to ASCII )
CR     ( -- , start new line , carriage return )
."     ( -- , output " delimited text )
```

Answers to Problems

If your answer doesn't exactly match these but it works, don't fret. In Forth, there are usually many ways to the same thing.

Stack Manipulations

```
1) SWAP DUP
2) ROT DROP
3) ROT DUP 3 PICK
4) SWAP OVER 3 PICK
5) -ROT 2DUP
```

Arithmetic

(12 * (20 - 17)) ==> `20 17 - 12 *`

(1 - (4 * (-18) / 6)) ==> `1 4 -18 * 6 / -`

(6 * 13) - (4 * 2 * 7) ==> `6 13 * 4 2 * 7 * -`

```
: SQUARE ( N -- N*N )
    DUP *
;
: DIFF.SQUARES ( A B -- A*A-B*B )
    SWAP SQUARE
    SWAP SQUARE -
;
: AVERAGE4 ( A B C D -- [A+B+C+D]/4 )
    + + +   ( add'em up )
    -2 ashift ( divide by four the fast way, or 4 / )
;
: HMS>SECONDS  ( HOURS MINUTES SECONDS -- TOTAL-SECONDS )
    -ROT SWAP ( -- seconds minutes hours )
    60 * +    ( -- seconds total-minutes )
    60 * +    ( -- seconds )
;
```

Chapter 5
Intermediate Forth Tutorial

Note: This chapter is shared between the JForth Manual and the HMSL manual. Information specific to HMSL on the Macintosh or JForth on the Amiga is noted as such.

Editing Programs in Files

As your programs get larger, it becomes impractical to type them in fresh each time. To develop software of any size, you'll need to keep your source code in files. You can then compile directly from these files instead of from the keyboard. Forth programs can be created like programs in other languages by using a text editor. You can use any text editor that you like. Beware of word processing programs because they will put strange information in the file related to margins, fonts, etc. If you use a word processor, save the file in *text only* mode. We recommend that you use the text editor that comes included with your Forth package. The process of editing differs between computers. Please read the appropriate section for your machine

Editing on the Macintosh: To create a new file, select New from the File menu while in Forth. Text can be enterred in the standard Macintosh style. Look in the Edit menu for commands like Cut, Copy and Paste, Upper and Lower case conversion.

Editing on the Amiga: The editor supplied with JForth is called Textra. It was written by Mike Haas using JForth. Full documentation on Textra is supplied in the JTX:DOCS directory. You will probably want to install Textra by either copying it to your C: directory (or any other directory in your current AmigaDOS path) or to a place convienient for double-clicking it's icon. Textra can be found in the JTX: directory on the JTOOLS: disk. Assuming Textra is installed, either double-click on it's icon or enter:

```
RUN TEXTRA
```

If you want to open an existing file, select it using the file selector. To start with a fresh file, hit CANCEL. Look in the Project menu for commands to save files and to create new files.

You can move through the file using cursor keys or the mouse. Drag the mouse (while holding down the left mouse button) to select text. Selected text can be Cut, Copied, or Pasted using commands in the Edit menu.

If you have ARexx, look in Chapter 23, *AREXX Support* for instructions on how to connect JForth and Textra for easy compilation.

Sample Program

Enter into your file, the following code.

```
\ Sample Forth Code
\ Author: your name
: SQUARE  ( n -- n*n , square number )
```

```
    DUP *
;
: TEST.SQUARE ( -- )
    CR ." 7 squared = "
    7 SQUARE . CR
;
```

Now save the file as described for your machine.

The text following the \ character is treated as a comment. This would be a REM statement in BASIC or a /*---*/ in 'C'. The text in parentheses is also a comment.

Save this file using the "Save As..." menu option, giving it the name SAMPLE in the process.

INCLUDE the Program

"INCLUDE" in Forth means to compile from a file.

Compiling on the Macintosh: To compile a file directly from the editor, select the "Include From Editor" command from the Include menu. If an error is encountered, it will be highlighted in the editor. Error messages will appear in the Forth window. You can also compile your program from HMSL by selecting "Include from File..." from the File menu. Use the dialog to select the file you just created.

Compiling on the Amiga: If you have ARexx installed, and configured Textra and JForth as described in Chapter 23, you can compile a file from Textra by simply hitting whichever function key is assigned to the jcompile command. If an error is encountered, it will be highlighted in the editor. Error messages will appear in the Forth window.

You can also compile from a file using the INCLUDE command. If you saved your file as WORK:SAMPLE, then compile it by entering:

```
INCLUDE WORK:SAMPLE
```

Forth will compile your file and tell you how many bytes it has added to the dictionary. To test your word, enter:

```
TEST.SQUARE
```

Your two words, SQUARE and TEST.SQUARE are now in the Forth dictionary. We can now do something that is very unusual in a programming language. We can "uncompile" the code by telling Forth to **FORGET** it. Enter:

```
FORGET SQUARE
```

This removes SQUARE and everything that follows it, ie. TEST.SQUARE, from the dictionary. If you now try to execute TEST.SQUARE it won't be found.

Now let's make some changes to our file and reload it. Go back into the editor and make the following changes: (1) Change TEST.SQUARE to use 15 instead of 7 then (2) Add this line right before the definition of SQUARE:

```
ANEW TASK-SAMPLE
```

Now Save your changes and go back to the Forth window.

You're probably wondering what the line starting with **ANEW** was for. ANEW is always used at the beginning of a file. It defines a special marker word in the dictionary before the code. The word

typically has "TASK-" as a prefix followed by the name of the file. When you ReInclude a file, ANEW will automatically FORGET the old code starting after the ANEW statement. This allows you to Include a file over and over again without having to manually FORGET the first word. If the code was not forgotten, the dictionary would eventually fill up.

If you have a big project that needs lots of files, you can have a file that will load all the files you need. Sometimes you need some code to be loaded that may already be loaded. The word **INCLUDE?** will only load code if it isn't already in the dictionary. In this next example, I assume the file is on the volume WORK: and called SAMPLE. If not, please substitute the actual name. Enter:

```
FORGET TASK-SAMPLE
INCLUDE? SQUARE WORK:SAMPLE
INCLUDE? SQUARE WORK:SAMPLE
```

Only the first INCLUDE? will result in the file being loaded.

Variables

Forth does not rely as heavily on the use of variables as other compiled languages. This is because values normally reside on the stack. There are situations, of course, where variables are required. To create a variable, use the word **VARIABLE** as follows:

```
VARIABLE MY-VAR
```

This created a variable named MY-VAR . A space in memory is now reserved to hold its 32-bit value. The word VARIABLE is what's known as a "defining word" since it creates new words in the dictionary. Now enter:

```
MY-VAR .
```

The number you see is the address, or location, of the memory that was reserved for MY-VAR. To store data into memory you use the word **!** , pronounced "store". It looks like an exclamation point, but to a Forth programmer it is the way to write 32-bit data to memory. To read the value contained in memory at a given address, use the Forth word @ , pronounced "fetch". Try entering the following:

```
513 MY-VAR !
MY-VAR @ .
```

This sets the variable MY-VAR to 513 , then reads the value back and prints it. The stack diagrams for these words follows:

```
@  ( address -- value , FETCH value FROM address in memory )

!  ( value address -- , STORE value TO address in memory )

VARIABLE  ( <name> -- , define a 4 byte memory storage location)
```

A handy word for checking the value of a variable is **?** , pronounced "question". Try entering:

```
MY-VAR ?
```

If ? wasn't defined, we could define it as:

```
: ? ( address -- , look at variable )
    @ .
;
```

Imagine you are writing a game and you want to keep track of the highest score. You could keep the highest score in a variable. When you reported a new score, you could check it aginst the highest score. Try entering this code in a file as described in the previous section:

```
VARIABLE HIGH-SCORE
: REPORT.SCORE  ( score -- , print out score )
    DUP CR ." Your Score = "   .   CR
    HIGH-SCORE @ MAX   ( calculate new high )
    DUP ." Highest Score = "   .   CR
    HIGH-SCORE !       ( update variable )
;
```

Save the file to disk, then compile this code using the INCLUDE word. Test your word as follows:

```
123 REPORT.SCORE
9845 REPORT.SCORE
534 REPORT.SCORE
```

The Forth words @ and ! work on 32-bit quantities. Some Forths are "16-bit" Forths. They fetch and store 16-bit quantities. Forth has some words that will work on 8 and 16-bit values. C@ and C! work for 8-bit bytes. The 'C' stands for "Character" since ASCII characters are 8-bit numbers. Use W@ and W! for 16-bit "Words."

Another useful word is **+!** , pronounced "plus store." It adds a value to a 32-bit value in memory. Try:

```
20 MY-VAR !
5 MY-VAR +!
MY-VAR @ .
```

Forth also provides some other words that are similar to VARIABLE. Look in the glossary for VALUE and ARRAY. Also look at the section on "local variables" which are variables which only exist on the stack while a Forth word is executing.

A word of warning about fetching and storing to memory: You have now learned enough about Forth to be dangerous. The operation of a computer is based on having the right numbers in the right place in memory. You now know how to write new numbers to any place in memory. Since an address is just a number, you could, but shouldn't, enter:

```
73 253000 ! ( Do NOT do this. )
```

The 253000 would be treated as an address and you would set that memory location to 73. I have no idea what will happen after that, maybe nothing. This would be like firing a rifle through the walls of your apartment building. You don't know who or what you are going to hit. Since you share memory with other programs including the operating system, you could easily cause the computer to behave strangely, even crash. Don't let this bother you too much, however. Crashing a computer, unlike crashing a car, does not hurt the computer. You just have to reboot. The worst that could happen is that if you crash while the computer is writing to a disk, you could lose a file. That's why we make backups. This same potential problem exists in any powerful language, not just Forth. This might be less likely in BASIC, however, because BASIC protects you from a lot of things, including the danger of writing powerful programs.

Another way to get into trouble is to do what's called an "odd address memory access." The 68000 processor arranges words and longwords, 16 and 32 bit numbers, on even addresses. If you do a @ or **!** , or **W@** or **W!** , to an odd address, the 68000 processor will take exception to this and try to abort.

Forth gives you some protection from this by trapping this exception and returning you to the OK prompt. If you really need to access data on an odd address, check out the words **ODD@** and **ODD!** in the glossary. **C@** and **C!** work fine on both odd and even addresses.

("Odd address errors" are not possible on Amiga's that use 68020 CPU's and later...they are made to perform this type of operation. However, if you have one of these newer CPU's, you should always be mindful of all the Amiga's which DO use 68000's and avoid doing such operations.)

The reason I am spending so much time on this is that if your program is crashing and getting those little pictures of bombs, you may likely be storing somewhere you shouldn't be.

Constants

If you have a number that is appearing often in your program, we recommend that you define it as a "constant." Enter:

```
128 CONSTANT MAX_CHARS
MAX_CHARS .
```

We just defined a word called MAX_CHARS that returns the value on the stack when it was defined. It cannot be changed unless you edit the program and recompile. Using **CONSTANT** can improve the readability of your programs and reduce some bugs. Imagine if you refer to the number 128 very often in your program, say 8 times. Then you decide to change this number to 256. If you globally change 128 to 256 you might change something you didn't intend to. If you change it by hand you might miss one, especially if your program occupies more than one file. Using CONSTANT will make it easy to change. The code that results is equally as fast and small as putting the numbers in directly. I recommend defining a constant for almost any number.

Logical Operators

These next two sections are concerned with decision making. This first section deals with answering questions like "Is this value too large?" or "Does the guess match the answer?". The answers to questions like these are either TRUE or FALSE. Forth uses a 0 to represent **FALSE** and a -1 to represent **TRUE**. TRUE and FALSE have been capitalized because they have been defined as Forth constants. Try entering:

```
23 71 = .
18 18 = .
```

You will notice that the first line printed a 0, or FALSE, and the second line a -1, or TRUE. The equal sign in Forth is used as a question, not a statement. It asks whether the top two items on the stack are equal. It does not set them equal. There are other questions that you can ask. Enter:

```
23 198 < .
23 198 > .
254 15 > .
```

In California, the drinking age for alcohol is 21. You could write a simple word now to help bartenders. Enter:

```
: DRINK? ( age -- flag , can this person drink? )
    20 >
;
```

```
20 DRINK? .
21 DRINK? .
43 DRINK? .
```

The word FLAG in the stack diagram above refers to a logical value.

Forth provides special words for comparing a number to 0. They are **0= 0>** and **0<** . Using 0> is faster than calling 0 and > separately. Enter:

```
23 0> .  ( print -1 )
-23 0> . ( print 0 )
23 0= .  ( print 0 )
```

For more complex decisions, you can use the *Boolean* operators **OR** , **AND** , and **NOT** . OR returns a TRUE if either one or both of the top two stack items are true.

```
TRUE TRUE OR .
TRUE FALSE OR .
FALSE FALSE OR .
```

AND only returns a TRUE if both of them are true.

```
TRUE TRUE AND .
TRUE FALSE AND .
```

NOT reverses the value of the flag on the stack. Enter:

```
TRUE .
TRUE NOT .
```

Logical operators can be combined.

```
56 3 >    56 123 < AND .
23 45 =   23 23 = OR .
```

Here are stack diagrams for some of these words. See the glossary for a more complete list.

```
< (  a b -- flag , flag is true if A is less than B )
> (  a b -- flag , flag is true if A is greater than B )
= (  a b -- flag , flag is true if A is equal to B )
0=  ( a -- flag , true if a equals zero )
OR  ( a b -- a||b , perform logical OR of bits in A and B )
AND  ( a b -- a&b , perform logical AND of bits in A and B )
NOT  ( flag -- opposite-flag , true if false, false if true )
```

Problem:

1) Write a word called LOWERCASE? that returns TRUE if the number on top of the stack is an ASCII lowercase character. An ASCII 'a' is 97 . An ASCII 'z' is 122 . Test using the characters " A a q z { ".

```
ASCII A LOWERCASE? .    ( should print 0 )
ASCII a LOWERCASE? .    ( should print -1 )
```

Flow of Control

You will now use the TRUE and FALSE flags you learned to generate in the last section. The "flow of control" words accept flags from the stack, and then possibly "branch" depending on the value. Enter the following code.

```
: .L ( flag -- , print logical value )
    IF ." True value on stack!"
    ELSE ." False value on stack!"
    THEN
;
0 .L
FALSE .L
TRUE .L
23 7 < .L
```

You can see that when a TRUE was on the stack, the first part got executed. If a FALSE was on the stack, then the first part was skipped, and the second part was executed. One thing you will find interesting is that if you enter:

```
23 .L
```

the value on the stack will be treated as true. The flow of control words consider any value that does not equal zero to be TRUE.

The **ELSE** word is optional in the **IF...THEN** construct. Try the following:

```
: BIGBUCKS? ( ammount -- )
    1000 >
    IF ." That's TOO expensive!"
    THEN
;
531 BIGBUCKS?
1021 BIGBUCKS?
```

Forth also has a **CASE** statement similar to switch() in 'C'. Enter:

```
: TESTCASE ( N -- , respond appropriately )
    CASE
        0 OF  ." Just a zero!" ENDOF
        1 OF  ." All is ONE!"  ENDOF
        2 OF  WORDS            ENDOF
        DUP . ." Invalid Input!"
    ENDCASE CR
;
0 TESTCASE
1 TESTCASE
5 TESTCASE
```

See CASE in the glossary for more information.

Problems:

1) Write a word called DEDUCT that subtracts a value from a variable containing your checking

account balance. Assume the balance is in dollars. Print the balance. Print a warning if the balance is negative.

```
VARIABLE  ACCOUNT
: DEDUCT ( n -- , subtract N from balance )
   ??????????????????????????????????  ( you fill  this in )
;
300 ACCOUNT !  ( initial funds )
40 DEDUCT  ( prints 260 )
200 DEDUCT ( print 60 )
100 DEDUCT  ( print -40 and give warning! )
```

Loops

Another useful pair of words is **BEGIN...UNTIL** . These are used to loop until a given condition is true. Try this:

```
: YAKYAK ( -- )
    BEGIN
        ." I could talk all day!" CR
        ?TERMINAL
    UNTIL
;
YAKYAK
```

This word will keep running until you hit a key on the keyboard. The word **?TERMINAL** will return TRUE if a key has been hit. (You could then immediately call KEY if you wish to know what key was hit.)

If you know how many times you want a loop to execute, you can use the **DO...LOOP** construct. Enter:

```
: SPELL
    ." ba"
    4 0 DO
        ." na"
    LOOP
;
```

This will print "ba" followed by four occurrences of "na". The ending value is placed on the stack before the beginning value. Be careful that you don't pass the values in reverse. Forth will go "the long way around" which could take awhile. The reason for this order is to make it easier to pass the loop count into a word on the stack. Consider the following word for doing character graphics. Enter:

```
: PLOT# ( n -- )
    0 DO
        ASCII -   EMIT
    LOOP CR
;
CR  9 PLOT#  37 PLOT#
```

If you want to access the loop counter you can use the word I . Here is a simple word that dumps numbers and their associated ASCII characters.

```
: .CHARS ( end start -- , dump characters )
    DO
        CR I . I EMIT
    LOOP CR
;
80 64 .CHARS
```

You could make this word safe by making sure that the parameters are in the right order. The word **-2SORT** will take two numbers and leave the smallest on top. Enter:

```
: SAFE.CHARS ( n1 n2 -- )
    -2SORT  ( put smallest on top )
    .CHARS
;
40 50 SAFE.CHARS
50 40 SAFE.CHARS
```

If you want to leave a DO LOOP before it finishes, you can use the word **LEAVE**. Enter:

```
: ANNOY.ME ( -- , Sing! )
    50000 0
    DO  50000 I - .
        ." bottles of beer on the wall." CR
        ?TERMINAL ( was a key hit )
        IF  ." OK, I'll shut up!" CR
            LEAVE  ( quit looping )
        THEN
    LOOP
;
```

Please consult the manual to learn about the following words **+LOOP** and **RETURN** .

Another useful looping construct is the **BEGIN WHILE REPEAT** loop. This allows you to make a test each time through the loop before you actually do something. The word WHILE will continue looping if the flag on the stack is True. Suppose you want to write a word called SHOWCHARS that prompts the user to hit a key, waits for a key to be hit, then prints the key as a decimal number and as a character. It should keep doing this in a loop until the key is a 'q' or a 'Q'. Here is a word that will do this:

```
: SHOWCHARS  ( -- , loop on chars till 'q' )
   BEGIN
      ." Hit a key or 'q' to quit:" KEY   ( wait for a key )
      DUP ASCII q =  ( -- char flag )
      OVER ASCII Q = OR NOT
   WHILE  ( -- char , loop if not 'q' or 'Q' )
      CR ." Key = " DUP . EMIT CR
   REPEAT DROP
;
```

Text I/O

You learned earlier how to do single character I/O. This section concentrates on using strings of characters. A text string in Forth consists of a character count in the first byte, followed immediately by the characters themselves. A character string can be created using the Forth word " , pronounced 'quote'. Note that you must follow the " by one space. Any text following that space is copied to a special area in memory called the **PAD**. The text string is terminated by an ending " . Enter:

```
" Fred" .
```

The number that was printed was the address of the start of the string. It should be a byte that contains the number of characters. Now enter:

```
" Fred" C@ .
```

You should see a 4 printed. Remember that C@ fetches one character/byte at the address on the stack. By adding one to this address and doing a C@, you should see the first character. Enter:

```
" Fred" 1+ DUP C@ EMIT
1+ DUP C@ EMIT
1+ C@ EMIT
```

Using this method, you could type out any string. Luckily, there are two words that make this process much easier. Enter:

```
" Hello" COUNT .S
TYPE
```

The word **COUNT** extracts the number of characters and their starting address. **TYPE** accepts the output of count and prints those characters. COUNT will only work with strings of less than 256 characters, since 255 is the largest number that can be stored in the count byte. TYPE will, however, work with longer strings since the length is on the stack. Their stack diagrams follow:

```
COUNT  ( $addr -- addr #bytes , extract string information )
TYPE   ( addr #bytes -- , output characters at addr )
```

The $addr is the address of a count byte. The dollar sign is often used to mark words that relate to strings.

You can easily input a string using the word **EXPECT**. (You may want to put these upcoming examples in a file since they are very handy.) The word EXPECT receives characters from the keyboard and places them at any specified address. EXPECT takes input characters until a maximum is reached or a carriage return is entered. The user variable SPAN contains the number of characters entered. You can write a word for entering text. Enter:

```
: INPUT$ ( -- $addr )
    PAD 1+  ( leave room for byte count )
    128 EXPECT  ( recieve a maximum of 128 chars )
    SPAN @   PAD C!  ( set byte count )
    PAD    ( return address of string )
;
```

You could use this in a program that writes form letters.

```
: FORM.LETTER ( -- )
    ." Enter customer's name." CR
```

```
        INPUT$
        CR ." Dear " DUP COUNT TYPE CR
        ." Your cup that says " COUNT TYPE
        ."  is in the mail!" CR
    ;
```

EXPECT (addr maxbytes -- , input text, save at address)

SPAN (-- addr , contains number of chars EXPECT got)

You can use your word INPUT$ to write a word that will read a number from the keyboard. Enter:

```
: INPUT# ( -- N true | false )
    INPUT$  ( get string )
    NUMBER? ( convert to a string if valid )
    IF DROP TRUE  ( get rid of high cell )
    ELSE FALSE
    THEN
;
```

This word will return a single-precision number and a TRUE, or it will just return FALSE. The word **NUMBER?** returns a double precision number if the input string contains a valid number. Double precision numbers are 64-bit so we DROP the top 32 bits to get a single-precision 32 bit number.

Changing Numeric Base

Our numbering system is decimal, or "base 10." This means that a number like 527 is equal to (5*100 + 2*10 + 7*1). The use of 10 for the numeric base is a completely arbitrary decision. It no doubt has something to do with the fact that most people have 10 fingers (including thumbs). The Babylonians used base 60, which is where we got saddled with the concept of 60 minutes in an hour. Computer hardware uses base 2, or "binary". A computer number like 1101 is equal to (1*8 + 1*4 + 0*2 + 1*1). If you add these up, you get 8+4+1=13 . A 10 in binary is (1*2 + 0*1), or 2. Likewise 10 in any base N is N .

Forth makes it very easy to explore different numeric bases because it can work in any base. Try entering the following:

```
DECIMAL  6   BINARY .
1 1 + .
1101 DECIMAL .
```

Another useful numeric base is *hexadecimal*. which is base 16. One problem with bases over 10 is that our normal numbering system only has digits 0 to 9. For hex numbers we use the letters A to F for the digits 10 to 15. Thus the hex number 3E7 is equal to (3*256 + 14*16 + 7*1). Try entering:

```
DECIMAL
12 .HEX
12 256 *    7 16 * +  10 + .S
DUP BINARY .   .HEX
```

A variable called **BASE** is used to keep track of the current numeric base. The words HEX , **DECIMAL** , and **BINARY** work by changing this variable. You can change the base to anything you want. Try:

```
DECIMAL
7 BASE !
6 1 + .
BASE @ .
```

You are now in base 7 . When you fetched and printed the value of BASE, it said 10 because 7, in base 7, is 10.

You could define a word like .HEX for any base. What is needed is a way to temporarily set the base while a number is printed, then restore it when we are through. Try the following word:

```
: .BINARY ( N -- , print N in Binary )
    BASE @      ( save current base )
    2 BASE !    ( set to binary )
    SWAP .      ( print number )
    BASE !      ( restore base )
;
DECIMAL
22 .BINARY
22 .
```

Answers to Problems

Logical Operators

```
: LOWERCASE? ( CHAR -- FLAG , true if lowercase )
    DUP 123 <
    SWAP 96 > AND
;
```

Flow of Control

```
: DEDUCT  ( n -- , subtract from account )
   ACCOUNT @   ( -- n acc )
   SWAP - DUP ACCOUNT !  ( -- acc' , update variable )
   ." Balance = $" DUP . CR  ( -- acc' )
   0<  ( are we broke? )
   IF    ." Warning!! Your account is overdrawn!" CR
   THEN
;
```

Chapter 6
Advanced Forth Tutorial

String Handling

Since we will be doing a lot of string output, let's define a very handy word.

```
: $. ( $addr -- , print string )
    COUNT TYPE
;
" Hello" $.
```

As you were experimenting with text in the last lesson, you may have noticed that your string on the PAD got overwritten. This is because the PAD is a popular place to put text in FORTH. If you want to keep your string intact, you will need to copy it to a safe place. To make room for a string you can use two new words called CREATE and ALLOT. Enter:

```
CREATE MY-STRING   130 ALLOT
```

CREATE defines a new word similar to the way VARIABLE does. ALLOT will make room for 130 bytes (characters) after MY-STRING. We will discuss these words in detail later. You can move a string from one location to another using $MOVE. We will use our INPUT$ word from the previous tutorial. Enter:

```
INPUT$     ( hit return then enter a string )
MY-STRING $MOVE
```

The string you entered is now stored at MY-STRING. The PAD can now be used by other words, like " , without destroying your word. Just to prove it, enter:

```
PAD $.
" Smash" $.
PAD $.
MY-STRING $.
```

Comparing two strings is a common task. Enter:

```
" FROG" MY-STRING $MOVE
" TREE" MY-STRING $= .
" FROG" MY-STRING $= .
```

The word $= is useful to check for string matches. This might find application in a security system. Enter:

```
: PASSWORD ( -- flag, check password )
    CR ." Enter password: " INPUT$
    MY-STRING $=  DUP NOT ( test )
    IF ." Invalid password!"
    THEN
;
```

```
PASSWORD .
```

If you want to alphabetize strings, you also need to know which string is "higher". Enter:

```
" AARDVARK" MY-STRING $- .
" FROG" MY-STRING     $- .
" ZEBRA" MY-STRING    $- .
" frog" MY-STRING     $- .
```

$- compares two strings. It is sensitive to upper and lower case. Use TEXT=? if you want to ignore case.

If you want to append one string onto another you can do so using $APPEND . Enter:

```
" tree" MY-STRING $MOVE
"  frog" COUNT MY-STRING $APPEND
MY-STRING $.
```

The following example demonstrates Text, DO LOOPs, and LEAVE . It is a handy word that searches for a byte, or character, and tells you its offset in a string.

```
VARIABLE BOFFSET
: SEARCHBYTE ( byte addr count -- offset | -1 )
    -1 BOFFSET !  ( set default answer )
    0
    DO  DUP I + C@  ( get byte )
        ( -- byte addr byte , stack looks like this )
        2 PICK =    ( matches? )
        IF I BOFFSET ! LEAVE  ( save offset )
        THEN
    LOOP    2DROP
    BOFFSET @  ( get result )
;
ASCII t " Look a tree!" COUNT .S
SEARCHBYTE .
```

Saving Forth

You may find that you are always INCLUDEing a certain set of files when you use JForth. Rather than INCLUDE them each time, you can INCLUDE them once and save the compiled result in another file. Insert your JForth disk and enter:

```
DOS EXECUTE JFORTH:ASSIGNS
INCLUDE JU:SQRT
49 SQRT .
```

Now put in your formatted JWORK: disk from the previous tutorial and enter:

```
SAVE-FORTH  JWORK:SQRT4TH   ( save entire Forth in file)
BYE     ( this leaves JForth )
RUN JWORK:SQRT4TH    ( enter in CLI window )
( wait for it to load )
49 SQRT . ( SQRT is now available immediately )
```

When you SAVE-FORTH , you also have the option of expanding the dictionary space available for use. When JForth boots, it looks at a VARIABLE called #K that tells it how much space to allocate. To allocate 20K more dictionary, try.

```
MAP  ( see how much is there now )
20 #K +!
SAVE-FORTH JWORK:BIGGER4TH  ( may need blank disk )
BYE
RUN JWORK:BIGGER4TH  ( from CLI as before )
MAP  ( there should be more room )
```

You may want to prepare a Forth image with your favorite tools loaded. You can also expand the amount of USER variable space available by using #U in the same way. See #U in the glossary for more information.

Programming Aids

Sometimes a program doesn't behave as expected. This, by definition, is a 'bug'. Luckily, with Forth you have more debugging tools at your disposal than with almost any other language. These tools can be embedded right in the language so they are always available. You can examine any variable or constant. You can also execute most low level words by themselves for incremental testing. We have included, in JForth, some of the tools that we have found handy.

We have already used WORDS. This causes all of the Forth words to be printed out to the screen. You can stop output by hitting a space bar. Once it is stopped, a line of Forth can be entered. The word listing will continue after that line finishes. Another word that is handy is WORDS-LIKE. If you can't remember the exact name of a word but think it had a + in it, you could enter:

```
WORDS-LIKE +
```

also try:

```
WORDS-LIKE EMIT
WORDS-LIKE $
```

Once you have found a word, you might want to see what the code looks like for it. We have a special word called FILE? that tells you what file any word was loaded from. It then asks you if you want to see the source code. As long as the source is not in the assembly language kernel, you will be able to see it. Enter:

```
FILE? <FASTEMIT>
```

Try loading in some of your code and then use FILE? on your words. FILE? works by scanning the file for occurrences of the name. When it finds one, it prints lines until it finds a blank line. This has the result of also showing you examples of how the word is called if it is referenced in that file.

If you are curious about how a word actually works, you can disassemble it using DEF. To fully understand what you are seeing, you should study the chapter on assembly language, and the chapter on the disassembler. Try entering:

```
: ADD3 3 + ;
DEF ADD3
: EMITA ASCII A EMIT ;
DEF EMITA
```

(A quick note to Forth/68000 experts. TOS is D7, which is our cache for the Top Of Stack. DSP is A6, which is our Parameter or Data Stack Pointer.)

The Forth Interpreter and Dictionary

This section will reveal a little of what makes Forth such a unique and powerful language. All Forths keep a list of their defined words in something called a "dictionary". This dictionary is what you see when you call WORDS . When you hit a carriage return, a program called the Forth "interpreter" reads each word on that line and looks it up in the dictionary. If it finds the word, it executes the code that is associated with it. If it doesn't find it, it flashes the screen, then reprints the word followed by a '?'.

You can perform this sequence of operations yourself. Enter:

```
: HI ." HELLO" CR ;
' HI .S
EXECUTE
```

The apostrophe is a Forth word pronounced "TICK". It takes the word that follows it and looks it up in the dictionary. If it finds the word, it leaves the address of that word's associated code on the stack. That address is referred to as a "CFA", pronounced "c f a". CFA stands for Code Field Address. The word EXECUTE takes that address and executes it.

If you attempt to tick a word that doesn't exist, you will get an error message. This is a handy way of finding out whether a word is defined.

```
' GADCZXW   \ gives error message
```

You can convert the address of the code, CFA, to the address of the name of a word, the NFA, using >NAME. This is pronounced "to name".

Enter:

```
' HI
>NAME ID.
```

ID. takes the address of the word's name and types the name out. You can't use COUNT and TYPE because some extras bits are set in the count byte for internal use.

The behavior of the word ' varies from one dialect of Forth to another. One of the few ways in which JForth deviates from the Forth 83 standard is illustrated as follows:

```
: SAYHI ' HI EXECUTE ;
SAYHI
```

In most Forths written before Forth 83, and JForth, the ' in SAYHI will compile the CFA of HI . In a true Forth 83 system, the ' would not run until you execute SAYHI. SAYHI would then be followed by the word you want to "TICK". See the MULTI-STANDARDS file for information on making JForth compatible with Forth 83.

If you do want to write a word that "TICKS" another word, you must use [COMPILE] . Try entering:

```
: DUMPSWAP  ( -- , just dump SWAP )
    ' SWAP
    >NAME 40 DUMP
;
DUMPSWAP
```

```
( This next word is more useful )
: DUMPWORD  ( <name> -- , dump 40 bytes of word )
    [COMPILE] '
    >NAME 40 DUMP
;
DUMPWORD SWAP
DUMPWORD HI
```

Every word in the Forth dictionary is linked to the previous word by its link field. We can find out what is before HI by converting its NAME field to its LINK field and seeing where it points. Enter:

```
' HI   >NAME   .S   ( get NFA )
N>LINK  .S   ( get Link Field Address LFA )
@  ID.   ( get NFA that it points to and print it. )
```

You may want to put these next words in a file. They illustrate techniques for threading through the dictionary. BACKWORDS can be used for listing words defined just prior to another word in the dictionary. Enter:

```
: LOOKBACK ( nfa1 -- nfa2 , show word, link )
     DUP ID. CR
     N>LINK @  ( LFA points to previous NFA)
;
: BACKWORDS  ( N <word> -- , show N previous words )
    20 MIN   ( do a maximum of 20 )
    [COMPILE] '  ( get CFA )
    >NAME   SWAP 0   ( convert to NFA )
    DO  LOOKBACK
        DUP 0=       ( check for end)
        IF LEAVE THEN
    LOOP  DROP
;
```

Include this code and test by entering:

```
10 BACKWORDS DUP
```

If you are interested in experimenting more, look at the definitions for NAME> , DEFER , INLINE and MAX-INLINE . Also look at the chapter on JForth Internals and Memory Organization.

Return Stack

Consider the following two words:

```
: BIRD  ." BIRD" ;
: PROFOUND ." The " BIRD ."  flies!" CR ;
PROFOUND
```

The word PROFOUND calls BIRD. We hope that when BIRD finishes, the word PROFOUND will continue to completion. Execution must "return" to PROFOUND when BIRD finishes. To do this the processor places the address where execution should resume on the "Return Stack." At the end of BIRD is a 68000 RTS instruction that pops that address off the Return Stack and branches there. The word R@ can be used to examine this stack. Let's use it to look at the return address. Enter:

```
: RLOOK  R@ . ;
: TEST RLOOK ." Hello" CR ;
TEST
' TEST 4 + >ABS .
' TEST 4 + EXECUTE
DEF TEST
```

The address you see printed is the address of the code that prints "Hello." This is what was to be executed after RLOOK. The 68000 uses absolute addresses on the return stack thus you had to convert your calculated relative JForth address using >ABS.

The return stack can be also be used for purposes other than storing return addresses. You can temporarily store values there, with care. The only rule is that the return stack must be restored to its original state before the word finishes. Otherwise the 68000 will try to return to the wrong place, with unpredictable results. Enter the following code:

```
: TESTR  ( N M -- N+1 M )
    >R   ( save M on return stack )
    1+   ( increment N )
    R>   ( get M back from return stack )
;
```

The word >R , pronounced "to r", moves the top of the data stack to the return stack. The word R> , pronounced "r from", moves data back from the return stack to the data stack. There must always be an equal number of calls to >R and R> in any given word or that word will return to the wrong place. Only use R> to get numbers that YOU placed there.

Words for manipulating the return stack are handy for avoiding excessive data stack manipulation and can result in slightly faster code.

The previous example could have been coded using swap, but would have been slower.

```
: TESTS  SWAP 1+ SWAP ;  ( slower than TESTR )
```

Look in the glossary for information on RPICK , R0 , RP@ , RP! , and RDROP .

Extending the Compiler

Words like VARIABLE and CONSTANT can be used to define new words. For this reason they are called "defining words".

Suppose you want to write a new defining word called INTEGER. You can make this data structure behave just like a CONSTANT . Although this will be redundant, it will, hopefully, illustrate the use of two important words, CREATE and DOES> . Enter:

```
: INTEGER    ( value <name> -- )
    CREATE    ,   ( save value in dictionary )
    DOES>     @   ( fetch when executed )
;
173 INTEGER MYINT     ( execute CREATE code )
MYINT .               ( execute DOES> code )
```

A word like INTEGER has two main parts. The first part is the code between CREATE and DOES> . This code executes when the defining word is executed, such as when MYINT is defined in the

preceding example. In this case the value on the stack is saved as part of MYINT. The comma takes what is on the stack and saves it in the dictionary.

The second part is the code that between DOES> and ';' . This describes what the newly defined word will do when executed. The address of a data area called the "BODY", or "PFA", is passed to the DOES> code. In the above example, the @ gets the value that was saved in MYINT at definition time.

In this next example, CREATE DOES> is used to describe a new kind of word called a SPEAKER. A SPEAKER is given a phrase to say whenever it is referenced. Enter the following code. (Remember the text in parentheses are comments and need not be entered.)

```
: SPEAKER    ( $string <name> -- )
    CREATE  HERE $MOVE    ( save string in dictionary )
      HERE C@ ALLOT ALIGN  ( adjust DP )
    DOES>   COUNT TYPE CR
;
" Billions and billions!" SPEAKER CARL
" Four score!" SPEAKER ABE  ( execute CREATE code )
CARL      ( execute DOES> code )
ABE
```

The DP that was referred to was the dictionary pointer. It is the address of where new code will be compiled, and should be left at an even location.

For another example of the use of CREATE DOES> , see the file JU:VALUE.

Earlier in the tutorial, we described the Forth interpreter as taking words from the input stream and executing them. This is clearly not the case when a new word is being compiled. When the Forth interpreter finds a colon, it switches to a different state, known as "compile mode". As words are input, instead of being immediately executed, they are compiled into whatever word is being defined. Now you obviously need a way to get out of this state and back to "interpet mode" at the end of a word. To accomplish this, Forth has what are called IMMEDIATE words that are executed immediately, even in compile mode. Semicolon, ';', is one such word. Semicolon finishes the definition of a word and switches the state back to interpret mode.

The state of the interpreter is stored in a VARIABLE called, appropriately enough, STATE. Enter:

```
STATE @ .
```

Since you are in interpret mode, you should see a 0. Let's make our own IMMEDIATE word so that we can spy on STATE in the middle of compiling. Enter:

```
: STATE.PEEK  ( -- , show state )
    ." State = " STATE @ . CR
;
IMMEDIATE ( Makes last word IMMEDIATE. )
STATE.PEEK
: FOO STATE.PEEK  ." Hello." CR ;
```

Forth has two other words which can change STATE. They allow you to switch back temporarily to interpret mode in the middle of a definition. Enter:

```
: BYTE.MASK    ( n -- byte )
    STATE.PEEK
    [ HEX STATE.PEEK ]  FF AND   [ DECIMAL ]
```

```
;
```

The left bracket sets STATE to 0 so that the following words are executed immediately. The right bracket sets STATE back to TRUE so that compilation can continue. This can be useful if you want to do a complex calculation at compile time instead of execution time. Compare the following two ways of defining the same word:

```
: DAYS2SECS    ( #days -- #seconds )
    24 * 60 * 60 *
;
DEF DAYS2SECS
( or )
: DAYS2SECS    ( #days -- #seconds )
    [ 24 60 * 60 * ] LITERAL ( calculate factor )
    *   ( only one multiply at compile time )
;
DEF DAYS2SECS
```

In the second case, the calculation of a single multiplication factor is done at compile time. The word LITERAL is used to compile that value into the dictionary as a constant. The use of this technique can result in faster code. The second word is equivalent to:

```
: DAYS2SECS    86400 * ;
```

If IMMEDIATE words always execute immediately whenever referenced, then how can one compile a call to an IMMEDIATE word? You can do this with [COMPILE] which you saw used with ' earlier. [COMPILE] will cause the next word to be compiled instead of executed. Suppose you wanted to write a word that had LITERAL compiled within it. Enter:

```
VARIABLE MCOUNT
10 MCOUNT !
: NEXTNUM  ( -- , Compile number and increment. )
    MCOUNT @ DUP ." MCOUNT = " . CR
    [COMPILE] LITERAL   ( don't execute LITERAL yet)
    1 MCOUNT +!
;
IMMEDIATE
: FOO  NEXTNUM + ;   ( 10 + ) ( execute LITERAL now )
: GOO  NEXTNUM + ;   ( 11 + )
5 FOO .
5 GOO .
5 FOO .
```

An associated word is COMPILE. Suppose we wanted to make a compiling word similar to the one above. Enter:

```
: +MC ( -- , Compile number and add )
    [COMPILE] NEXTNUM  ( it is immediate )
    COMPILE + ( compile a call to + when +MC is used)
;
IMMEDIATE
: ZOO +MC ;
5 ZOO .
```

```
5 FOO .
```

These words are difficult to explain. The best way to get a feel for them is to turn on TRAPS and start experimenting.

These techniques can be used for making new "flow of control" words. The words IF , THEN , BEGIN , UNTIL , etc., are all IMMEDIATE words. They must set up complex branches at compile time. Suppose you want to make a new conditional construct that only stops looping if a key is hit. Enter:

```
: UNTIL-KEY
    COMPILE ?TERMINAL
    [COMPILE] UNTIL
;
IMMEDIATE
: TESTIT
    BEGIN
        ." BLAH BLAH" CR
    UNTIL-KEY
;
TESTIT
```

We are really delving into the guts of Forth with these words. Don't feel bad if this stuff confuses you. It confuses me. Luckily these techniques are only needed for extending the language. Useful and productive lives can be led without them.

Further Exploration

There are a number of useful words that are part of standard Forth that were not covered in this tutorial. Look these up in the glossary and experiment with them on your own. You should now have the necessary skills to do so. Some examples are:

```
DEPTH    CMOVE   FILL    */    U<
U.       FIND    TIB     .R
```

I would also recommend reading the sections on Local Variables, Vocabularies, and Floating Point. When you want to get into the Amiga internals, there is a chapter on Accessing the Amiga. You will probably want to read the section on the demo programs. Print these programs and study them. They are your gateway to the Amiga's special capabilities.

Chapter 7
Clone - The JForth Target Compiler

Historically, Forth has provided a strong development environment; fast to write code in and easy to debug. Its natural interactivity strongly contributed to both features. However, Forth has been somewhat limited in its usefulness to the personal computer user, as it does not lend itself well to creating small standalone programs.

The main reason is that Forth has to carry around its dictionary, and usually much of it is never used by the application. This includes the Forth compiler itself, which is almost always disabled by vendor-provided turnkey programs! Other language components that standalone programs often tote along without using include debugging aids, INTERPRET-related words, assemblers, disassemblers, etc.

This is where CLONE comes in, and it really is something quite unique in the Forth world! We are very proud of CLONE and pleased to provide you this sophisticated development tool.

CLONE creates for you *another executable copy* of your compiled JForth program, but does so *outside* of the JForth dictionary. And as it builds this "clone", it *includes only the parts of JForth that are needed by your program!* This greatly reduces the size of your final program.

Also, executables created by CLONE are *YOURS!* You may do with them as you wish...*ROYALTY FREE!* This is an important consideration for commercial projects, as you are not allowed to distribute a JForth image (those with a dictionary) unless it has been processed by JU:TURNKEY to no longer be interactive.

How To Use Clone

1. Compile the CLONE program...
   a. From CLI or Shell, enter: Run Com:JForth
   b. From JForth, enter: INCLUDE CL:TOPFILE

Clone consumes approximately 30K of dictionary. (We recommend SAVE-FORTHing an image with CLONE compiled so that this step need not be repeated each time.)

2. Compile the program to be CLONEd.
   a. You may have to increase the dictionary size (via #K) and SAVE-FORTH to have enough room.
3. Enter: CLONE <NAME>

   where: <NAME> is the name of the main entry point of your program. Wait for "ok".
4. Enter: SAVE-IMAGE <NAME> <FILENAME> [-s -m -icon]

   where: <NAME> is the name of the main entry point of your program.

   <FILENAME> is the path/name of the executable file

options:

-s Include a debug symbol table for Wack

-m Write a "filename.map" file (this can be big)

-icon Create a "filename.info" icon file

Notes:

Executables created by CLONE may be launched from CLI or Workbench.

The above mentioned main entry point should *not* accept parameters on the stack, but rather in the command line, like CLI commands. These are available immediately when your program is started; the usual parsing words like WORD function normally to retrieve this information; the next WORD or LWORD operation moves the next argument on the command line to HERE.

Technical Information About Clone

CLONE is an expert at analyzing compiled JForth code. Given a main entry point, it derives a complete call dependency tree sufficient to rebuild a separate executable image of minimum size, including only those words necessary to support the run time environment of the main program.

In the process, the NAME-FIELD, LINK-FIELD, and SIZE-FIELD components are also filtered out, also reducing the size of the resultant executable. The standard JForth Demos image (loadable from the JD: directory of your release disks) shrank from about 160K, in the JForth dictionary, to less than 28K after passing through CLONE. No demo source file was altered from the original V1.2 release.

MINIMUM TARGET SIZE

The minimum size of a CLONEd JForth program is about 3 Kbytes. This is the code required to support both launching from CLI and Workbench environments. This can be verified by CLONEing the resident JForth word NOOP with the NOCONSOLE variable (described below) turned ON.

SPECIAL NOTE ON JFORTH "STATE"...

Since CLONE depends heavily on the resident dictionary to function as a model for the final program, the state of the JForth environment during CLONE is crucial. For example, if you type the JForth word SLOW before CLONEing (causing JForth to use unbuffered input/output to the console window) the CLONEd program will also exhibit this behavior.

Another example of how the state of JForth directly affects the output file is in the use of the compiler *MAX-INLINE* variable; whatever its setting when the program was compiled will affect the size of the created target.

Clone Glossary

These words form the user interface for the CLONE program. They are usually typed at the keyboard to create your standalone program...

CLONE (-- , <name>)

A two-pass program which first creates a call dependency tree for <*name*>, then uses the tree to build a separate program image in memory, of minimum size.

STATS (--)

Prints out a status report for the previous CLONE operation. This includes the name of each included routine, its address in both the normal dictionary and the new image, and the number of times it is referenced by other routines in the new image. (This same information is saved in the '.map' file if the *-m* option is included on the CLONE command line.)

SAVE-IMAGE (-- , <name> <filename> [<-s -m -icon>])

Creates an executable file on the disk for <name>, which must exist somewhere within the current call dependency tree. Accepts optional parameters. For more details, see *HOW TO USE CLONE*, above.

SHOWME (-- , <name>)

Once CLONE has been performed, disassembles the CLONEd code for <name>, using target-image-relative addressing.

INITCLONE (--)

Re-Initializes CLONE, freeing all CLONE-related resources and allowing another CLONE operation to be performed. Normally done between CLONE evolutions.

Two words are available to your program for special "cancel" behavior (applicable only if run fromCLI or Shell):

CANCELKEY? (-- n1)

Returns an ascii C, D, E, or F if the corresponding control key has been pressed at the keyboard; 0 otherwise. See ENABLE_CANCEL in the section "Customizing the CLONEd Image".

Prior to compiling the CLONE program, CANCELKEY? will unconditionally return 0.

CANCELNOW? (--)

If ENABLE_CANCEL has been set non-zero, calls CANCELKEY? to check if CTRL-C, D, E, or F has been pressed at the keyboard. If so, the image is exited; otherwise, no action occurs. See ENABLE_CANCEL in the section *Customizing the CLONEd Image*.

Prior to compiling the CLONE program, CANCELNOW? executes NOOP.

Customizing the CLONEd Image

Several words, mostly state variables, are available to the programmer to control CLONE's key run time behavior. Default values are employed that satisfy most programs, but for best results, each should be set or verified just prior to a CLONE operation. Refer ahead to the section *Clone Configuration File*, for an example of how this is done conveniently.

TRACKING (-- addr)

This is a variable, used as a boolean. If non-zero, CLONE will include all the normal memory and file pointer tracking mechanisms that are used in the JForth development image; this feature automatically returns still-allocated memory, and closes still-opened files and standard Amiga

libraries when the image is exited, but can increase the size of the executable and affect the run-time performance of those sections of the program which are memory allocation/deallocation intensive.

If zero, CLONE will not include any of this support, leaving the total responsibility for memory, files and libraries to the application. Well-behaved, debugged programs are always meticulous in these respects; such extra housekeeping is unnecessary (and arguably undesirable).

The default state for TRACKING is OFF.

NOCONSOLE (-- addr)

Another boolean variable, this controls whether I/O support for a normal AmigaDOS CLI window should be included. If your program relies on the standard Forth KEY, EMIT, ?TERMINAL, and/or TYPE words (including ."), you should set this variable OFF. This is the configuration for CLI-oriented commands like DIR, regardless of what kind of console window (RAW:, CON: or NEWCON:) is used.

On the other hand, if your program does not directly call any of the Forth I/O routines (such as a graphics-intensive game), NOCONSOLE should be turned ON. This will exclude the run-time code for the normal I/O primitives and yield a smaller executable.

The default state for NOCONSOLE is OFF.

ENABLE_CANCEL (-- addr)

This boolean variable matters only to CLI-based programs and, when set non-zero, causes CLONEd programs to automatically exit via QUIT on the first call to KEY after CTRL-C, D, E or F has been typed by the user. See the description of CANCELNOW? in the Clone Glossary section

ENABLE_CANCEL must be set non-zero BEFORE CLONEing if this ability is desired in the CLONEd program.

The default state for ENABLE_CANCEL is OFF.

RAWEXPECTECHO (-- addr)

Another boolean variable, used to adjust for differences in behavior between the 2 types of CLI-based Amiga windows, RAW: and CON: (also NEWCON:, after WorkBench 1.2). Should be set ON if your program opens a RAW: window; this causes EXPECT to echo typed characters. Characters are automatically echoed by a CON: or NEWCON: window, so for these two types, RAWEXPECTECHO should be turned OFF.

The default state for RAWEXPECTECHO is OFF.

IFLEAVELONG (-- addr)

This variable should be set ON if you expect your CLONEd image to exceed 128K, OFF otherwise. If an image is larger than 128K it may need to use long relocatable JSRs which need four bytes instead of the normal two bytes for short register relative JSRs. If CLONE finds that this variable needs to be set, it will abort the current CLONE operation and inform you. To recover from this error, enter:

```
INITCLONE
IFLEAVELONG ON
```

Then try the CLONE operation again.

The default state for IFLEAVELONG is OFF.

IFLONGBRANCH (-- addr)

This variable should be set ON if CLONE aborts and tells you to set it. To recover from this error, enter:

```
INITCLONE
IFLONGBRANCH ON
```

Then try the CLONE operation again.

The default state for IFLONGBRANCH is OFF.

STACKSIZE (-- addr)

This variable is examined by CLONE to determine the initial size for the DATA stack, allocated when the image is run. Forth programs are normally conservative in the use of the data stack; 4K is usually quite sufficient.

The default setting for STACKSIZE is 4096.

NOTE: This should not be confused with the setting of the "stack" in the WorkBench 'info' menu command. That parameter determines the size of the JForth return stack (AmigaDOS defaults this value to 4096, also; this satisfies most JForth programs here, too).

DICTIONARYSIZE (-- addr)

This variable is examined to determine the size of the workspace at HERE in the target image. As in most Forth memory maps, HERE returns the address where compiled code ends, and marks the first available free address of the dictionary. Even though the concept of a dictionary does not exist in CLONEd programs, the area is commonly referenced as a scratch string area, so SAVE-IMAGE adds "DictionarySize @'" number of bytes to the code segment as a small workspace. It should be noted that the specified area actually resides in the saved executable, so programs should allocate larger workspaces rather than using the area above HERE.

The default setting for DICTIONARYSIZE is 256, allowing 128 bytes for PAD, and another 128 bytes below that, shared by HERE and the number formatter. This default size represents a minimum.

INITIALIMAGESIZE (-- addr)

This variable becomes important when you are CLONEing with a minimum of free memory available. Before CLONEing, if you set INITIALIMAGESIZE to slightly (1 or 2 kbytes) more than the final image size, the amount of memory needed for CLONE to complete will be reduced by as much as 50%.

For example, if your CLONEd application size normally ends up at about 33k, and your last CLONE failed due to insufficient memory, enter:

```
INITCLONE  \ to free up memory and allow CLONE to re-init
DECIMAL 35 1024 * INITIALIMAGESIZE ! \ slightly more than needed
```

Try CLONEing again; much less memory should be needed this time. Of course, you may be just plain too low or fragmented for this to help (if so, reboot the Amiga to unfragment).

Please note that one of the first things CLONE does is to read INITIALIMAGESIZE and try to allocate that much memory. Therefore, if you set it to something prohibitively high, CLONE will seem to fail immediately due to insufficient memory.

By default, INITIALIMAGESIZE is set to 4096.

ERRORCLEANUP (--)

An optional DEFERed word allows the programmer to specify one cleanup word that will be executed IF the CLONEd program terminates via **QUIT** or **ABORT** (usually due to a fatal error). If used, it should return all Amiga resources that have been allocated (close files, free memory etc.), but *should take care not to free something already freed.* The stack diagram for a word placed into ERRORCLEANUP should be (--).

ERRORCLEANUP is not called if the application terminates normally.

By default, ERRORCLEANUP is set up to execute NOOP. *THE APPLICATION MUST SET THIS VECTOR AT RUNTIME* by including a line similar to the following in its initialization code...

```
' MyErrorCleanup is ErrorCleanup
```

USERCLEANUP (--)

An optional DEFERed word allows the programmer to specify one cleanup word that will be executed when the CLONEd program terminates **normally**. If used, it should return all Amiga resources that have been allocated (close files, free memory etc.), but *should take care not to free something already freed.* The stack diagram for a word placed into USERCLEANUP should be (--).

USERCLEANUP is not called if the application terminates via QUIT or by ABORTing.

By default, USERCLEANUP is set up to execute NOOP. *THE APPLICATION MUST SET THIS VECTOR AT RUNTIME* by including a line similar to the following in its initialization code...

```
' MyUserCleanup is UserCleanup
```

Clone Configuration File

One convenient, transparent method of configuring the clone variables is to include something like the following at the end of the main load file for your program:

```
EXISTS? clone    \ are we compiling on top of CLONE?
.IF              \ YES, set CLONE how I want it.
    include CLONE.CONFIG
.THEN
```

A sample CLONE.CONFIG follows:

```
\ Sample CLONE.CONFIG file  (set CLONE run-time behavior)
\
\ Keep in working directory, conditionally INCLUDE
\ from load file

decimal

Tracking      off
NoConsole     off
Enable_Cancel off
RawExpectEcho off
```

```
IfLeaveLong   off
4096 StackSize !
 256 DictionarySize !
4096 InitialImageSize !
' noop is ErrorCleanUp
' noop is UserCleanUp
```

Word Redefinitions under Clone

Some JForth words functionally change due to the standalone nature of the targeted program. Some are described here; see the file CL:REDEFS.F for a complete listing.

QUIT - terminates the image.

INTERPRET - a null word, does nothing. (No Dictionary possible)

?PAUSE - again, a null word...the CLI provides 'pause'.

WORD - preserves case, like LWORD.

Also, since there are no name fields in a CLONEd image, there can be niether dictionary nor reason to CLONE words that operate on such. Examples of such words include FIND VOCABULARY ORDER DEF DISM WORDS VLIST, etc. Most have been rendered ineffectual in CLONEd images, others could cause problems. Don't worry, though. It is actually quite hard to accidentally include these in your CLONEd program.

Note: you are *not permitted* to include any JForth *compiler primitive* or other such *code generating utility or program* in your standalone image, via CLONE or any other method or tool. This includes words such as CFA, CREATE : ASM CODE etc. Actually, these have been specifically written to not be CLONE-compatible, so they will not function correctly in a standalone image anyway.

How To Be Clone Compatible

Programs destined to be CLONEd should follow these guidelines:

ODE

An ODE program that is to be cloned must call OB.INIT at the beginning to set up the object stack and to initialize the dynamic object tracking. It must also load the file JO:CLONE_SUPPORT if not already loaded. See the chapter on ODE for more information.

Use Supported Data Structures

Use the system tools wherever possible. VARIABLE, ARRAY, CREATE, and DOES> are all supported data structures that will be faithfully reproduced in the final image both in data content and functionality.

The DEFER and GLOBAL-DEFER words are the only currently supported means of vectoring execution. An example CLONE-able execution array is provided below.

Runtime Initialization of Data-Storage Elements which contain Addresses

CLONE will automatically relocate the contents of DEFERed and GLOBAL-DEFERed words, as well as compiled address references created with ' <pronounced tick> or ALITERAL. Use of these tools insures CLONE-compatibility.

Similarly, the addresses returned by CREATE-DOES> children (such as VARIABLEs or ARRAYs) will also reflect the new addresses of their data in the CLONEd image, however, the contents of their data area will be unmodified (since they often do not contain addresses). This means that if you save addresses in such elements, your program must initialize them when it runs, before using them. This is accomplished simply by compiling in a statement like the following (assuming a VARIABLE is being used):

```
' MyWord MyVariable !  \ initialize MyVariable to point to
MyWord's CFA
```

For more example code, see the sample execution array later in this document.

Do not assign addresses as CONSTANTs. CONSTANT values are not changeable and therefore can't be re-initialized later by your program. Of course, if the address can never change (such as the address of a hardware device), then use of CONSTANT is appropriate as CLONE should not relocate them.

All STRUCT elements are considered by CLONE to be data-storage and those that hold addresses (even if defined as APTRs) should be initialized by the program.

Assembly Code

Be especially careful when using either assembler format when writing CLONE-compatible programs; the address generation problem noted above is easy to create. Always use the function as would the compiler to generate its CFA or data address at run time. For example, if you want to load the relative address of a variable called MYVARIABLE into A0:

WRONG, MyVariable's data addr is built at "assembly-time" and "hard-coded"...

```
( RPN Example )
    MyVariable #   ar0 an   long   move
( Forward ASM example )
    move.l  #[MyVariable],a0
```

CORRECT, 'MyVariable' is "asked" at run-time for its addr, then moved to a0..

```
( RPN Example )
    ] MyVariable [              \ pushes tos, puts addr in tos
    tos dn   0ar an  move       \ move it to A0
    dsp a@+  tos dn  move       \ restore original TOS
( Forward ASM example )
    callcfa  MyVariable          \ pushes TOS, puts addr into TOS
    move.l   tos,a0              \ move it to A0
    move.l   (dsp)+,tos          \ restore original TOS
```

Since the compiler is temporarily invoked to build the reference to MYVARIABLE, CLONE will later be able to relocate that code segment.

Run Time Initialized Example Execution Array

An example execution array:

```
3 ARRAY EXECARRAY        \ 3 possible executable words
```

```
: INIT-ARRAY   ( -- , set up addresses in exec array )
  ' NOOP       0 EXECARRAY !    \ put 'NOOP' cfa in 1st elmnt
  ' OPEN.ALL   1 EXECARRAY !    \ put 'OPEN.ALL' cfa in 2nd elmnt
  ' CLOSE.ALL  2 EXECARRAY !    \ put 'CLOSE.ALL' cfa in 3rd elmnt
;

: EXECUTE-FROM-ARRAY  ( #element -- , fetch and execute )
  EXECARRAY @EXECUTE
;
```

This program will work as long as INIT-ARRAY is called in the main program before the array is used.

Differences Between Original and Cloned Code

Small but distinct differences may be noted between the dictionary and CLONEd versions of programs, primarily due to optimizations that CLONE performs.

1. While VARIABLE references were compiled inline in JForth V1.2, the compiler in later versions will create CALLs to the VARIABLE's cfa in the normal dictionary. CLONE, however, converts them to inline in the generated image, reclaiming the speed advantage.
2. All occurences of USER variables are converted to the VARIABLE data structure. Functionally, there is no difference in JForth programs, and the VARIABLE construct is faster.
3. The target compiler will alter the particular manner that another function was called as appropriate based on the new target address. For example, if TEST1 called TEST2 via a JSR absolute in the original dictionary, CLONE will change it to a more efficient and smaller relative reference if possible in the new image.
4. All VERIFY-LIBS error checking is stripped from calls to Amiga libraries.
5. If the IFLONGBRANCH variable is set TRUE, all short ,8 bit, branch displacements are converted to long, 16 bit.

Chapter 8
File I/O

This chapter describes the JForth words that provide a simple, complete interface to the AmigaDOS file routines. A file is simply a collection of bytes that can be stored on a disk or in memory. Any kind of data can be stored in a file. The bytes may be ASCII characters as in a document or a program's source code. They could also be data from an experiment or binary code for execution by the computer.

As you might expect, support is provided to perform the standard file operations. These are:

1. Opening a file for use and getting a "file pointer" that is used when referring to a file.
2. Using the file-pointer to read the data, write new data and/or reposition yourself in the file, *seek*.
3. Closing the file, again using the identifying file-pointer, when finished.

Some additional tools have been provided to round out the stock supply of file utilities. These may not be needed in a typical application. These involve the generation of NUL terminated file names which Amiga DOS requires internally, and fast buffered I/O.

Note that the Amiga OS already provides interfaces to some types of Amiga system files, such as icons (those that end in *.info*), libraries and others. Such files are not normally accessed with these functions, but with the Amiga-supplied ones.

Before we get into too much detail, let's explore some of these tools in a tutorial.

File I/O Tutorial

When entering this tutorial be sure to enter it exactly as written, especially when reading files.. Otherwise you could overwrite memory causing a harmless but annoying crash.

Creating a Text File

Let's create a new file. We can store some text in the file then read it back out. Rather than create a file on floppy disk, let's make one on the RAM: disk. Enter:

```
VARIABLE MYFILE
NEW FOPEN RAM:FILE1 .S
MYFILE !
```

We just created a NEW file called "FILE1" on the volume RAM:. The number that was returned by FOPEN is a pointer to a special structure that we can use to access that file. We don't have to worry about what is in that structure. Just consider it as a unique identifier for that file. We can have multiple files open and refer to each of them by their file pointer. We saved the file pointer in a variable called MYFILE because we will need it later.

It is possible that your RAM: disk was full which can happen if you are low on memory. If so, FOPEN would have returned a ZERO for a file pointer. It is important to check to make sure that the file

pointer is not ZERO before proceeding. That is why we used .S to show the pointer. If you got a zero from FOPEN, try again using a formatted disk in DF1:, with a filename of "DF1:FILE1".

Now let's write to the file. Enter:

```
MYFILE @  ( get the file pointer )
 " Important Information" COUNT .S
FWRITE .
```

FWRITE expects a file pointer followed by an address and count. It writes the string to the file and then returns the number of characters written. If there is an error, it will return a -1.

If we are going to write more lines to the file, we should separate them with and "End Of Line" character. EOL is a constant equal to the character used to separate lines in a file. On the Amiga, this is an ASCII Linefeed. If we write again to the file, the new data will go right after the previous data. This is because file I/O uses an imaginary cursor that points into the file. This type of I/O is called "sequential I/O" because bytes are read or written one after the other in sequence. Enter:

```
MYFILE @  EOL  FEMIT
```

FEMIT is a handy word that uses FWRITE to write a single character to a file. We can now write another line to the file.

```
MYFILE @  " COST = 23"  COUNT  FWRITE .
MYFILE @  EOL  FEMIT
```

When we are finished with a file we should close it. Enter:

```
MYFILE @  FCLOSE
```

We have now opened a file, written data to it, and closed it. We can see the result of our work by entering in the CLI window:

```
TYPE  RAM:FILE1
```

or in JForth:

```
TYPEFILE RAM:FILE1
```

Reading a Text File

Now let's open that file, and read what we wrote. Enter:

```
FOPEN RAM:FILE1 .S
MYFILE !
```

We don't need to say NEW because we are opening an existing file. Now let's read the first 8 characters from the file. Enter:

```
PAD 200 ERASE  ( clear PAD )
MYFILE @  PAD 8  FREAD .
PAD 8 TYPE
```

We should see the number 8 printed after the FREAD which is the number of characters read. The data was stored at PAD which we saw using TYPE . Let's now read the rest of the file. Reading a file uses a cursor just like when writing. We are now positioned after the 8th character and can read from that point. Enter:

```
MYFILE @  PAD 100 FREAD .
```

Notice that the number printed was less than 100. The number reflects the actual number of bytes read. Since we reached the end of the file, we got fewer bytes than we asked for. This is one way to tell when you reach the end of a file. We can look at our data by entering:

```
PAD 30 DUMP
```

Now let's close our file. Enter:

```
MYFILE @  FCLOSE
```

JForth provides several tools that simplify reading text files. These include READLINE and DOLINES which will be discussed later.

Using Binary Data Files

We can also use files to store numbers in the form of binary data. In fact, anything in memory, arrays, structures, parts of the dictionary, whatever, can be written to a file using FWRITE. Let's create an array of numbers then store them in a file. We should put a count of how many numbers there are at the beginning of the file so we know how to read it later. First let's make an array of data to use. Enter:

```
CREATE MYDATA 123 ,  2931 ,  7 , 99712 ,  49 ,
VARIABLE NUM-ITEMS
5 NUM-ITEMS !
```

Now let's make a file to store this data in. Enter:

```
NEW  FOPEN  RAM:BDATA .S
MYFILE !
```

At the beginning of the file we should store the number of 4 byte data cells that will follow. This will help us later when we want to read the file. Enter:

```
MYFILE @  NUM-ITEMS  4  FWRITE .
```

This wrote the 4 bytes at the address NUM-ITEMS to the beginning of the file. In other words, we just wrote the contents of the variable NUM-ITEMS to the file. Now let's write the data.

```
MYFILE @  MYDATA  NUM-ITEMS @ CELLS  .S
FWRITE .
```

Each number in MYDATA occupies 4 bytes or 1 cell. By calling CELLS we calculate how many bytes the table of numbers occupies.

Rather than close the file and reopen it, let's just reposition ourselves to the beginning and start reading. The word FSEEK will move our cursor to anyplace in the file. We can move to a location relative to our current position, or relative to the beginning or end. Let's move to the beginning of the file.

```
MYFILE @  0  OFFSET_BEGINNING FSEEK .
```

The number printed was our old position in the file. (You can move zero bytes relative to your current position to find out where you are!) We can now read the number of data items in the file. Enter:

```
0 NUM-ITEMS !
NUM-ITEMS ?
MYFILE @  NUM-ITEMS 4 FREAD .
NUM-ITEMS ?
```

NUM-ITEMS now contains the number of data items in the file. Let's read the data. Enter:

```
MYFILE @  PAD  NUM-ITEMS @ CELLS FREAD .
PAD @ .  ( print 123 )
PAD 8 + @ .  ( print 7 )
```

The data is now stored on the PAD.

Sometimes, a data file can be so big that we don't want to load the whole thing into memory. You can write a word that will read randomly from a given position in a file. This word will check for errors when seeking. FSEEK will return a -1 if you have an error. A common error is tryin to go outside the bounds of the file. You may want to enter this example in a file for future use. Enter:

```
: GRABDATA  ( item# -- item , read an item )
\ Calculate offset, skipping count at beginning.
    CELLS CELL+
\ Position cursor in file.
    MYFILE @  SWAP OFFSET_BEGINNING  FSEEK
    0<  ABORT"  File Seek Failed!"
\ Read the number.
    MYFILE @  PAD 4 FREAD
    4 = NOT ABORT" File Read Failed!"
    PAD @
;
0 GRABDATA .  ( print 123 , the first item is # 0 )
3 GRABDATA .  ( print 99712 )
70 GRABDATA .  ( should report failure )
```

Now close the file. Enter:

```
MYFILE @  FCLOSE
```

This demonstrated the use of a simple binary data file. Very complex files, like the IFF files can also be accessed with these techniques. See the JIFF:IFF_SUPPORT or JU:SHOWHUNKS for more examples.

File I/O Reference

Opening Files

Prior to reading from or writing to a file, it must be 'opened'. JForth provides four words concerned with opening files.

FILEWORD (<filename> -- $addr , parse file name from input)

If you have a file that has spaces in the name, then you cannot use WORD to get the filename because it will only get the first word up to the space. FILEWORD will check to see if the first letter of a filename is a ", if so it will parse up to the next " for the end of the name. This name can then be passed to words that use $FOPEN.

```
: TESTF  ( -- ) FILEWORD COUNT TYPE ;
TESTF  mydata
TESTF "name with spaces"  ( this will work!)
```

FOPEN (<filename> -- file-pointer | false , opens file)

This reads the filename from the input stream using FILEWORD, opens the file and returns a pointer to a file control structure.

```
FOPEN  DF1:DATAFILE
```

If you do not specify a pathname, Amiga DOS will default to the current directory as set by the CD command.

0FOPEN (0name -- file-pointer | false , opens file)

In this case the filename is a NUL terminated string passed on the stack.

```
0" DF0:THISFILE"  0FOPEN
```

$FOPEN ($name -- file-pointer | false , opens file)

This accepts a standard Forth string, with a count byte. You should use FILEWORD instead of word if you want to get a filename from input.

```
" DF0:THISFILE"  $FOPEN
```

If the file could not be opened, these words return false.

Files are normally opened as existing, read/write. This means that the file specified must exist, its contents will be preserved across the open, and both read and write operations are allowed.

Another word, NEW (--), may precede the 'FOPEN' word to create a new file, or clear the contents of an existing file.

The 'FOPEN' words access a variable called FILEMODE to determine the desired mode for opening. NEW places the value MODE_NEWFILE there, OLD replaces it with MODE_OLDFILE. Note that, at the end of every 'FOPEN' operation, the mode will be reset to OLD .

NOTE: Do NOT execute NEW unless it is immediately followed by the open' operation. It's UNNERVING to clear a wanted file on open, just because you executed NEW and forgot about it!

It is important to check the results of a file being opened because errors can easily occur. Open errors are typically due to the file not being found because the name is incorrect or you are in the wrong directory.

```
: OPENFILE ( <name> -- , open a file )
     FOPEN  ( gets name from input stream )
     DUP \ save the file-pointer in a variable
     IF   MYFILE !
     ELSE CR ." File could not be opened!"  QUIT
     THEN
;
```

Reading and Writing to files.

The words supplied in JForth are a high-level interface to the AmigaDOS calls READ, WRITE, and SEEK. One minor difference in their use is that all addresses passed as parameters are relative addresses. They are converted to absolute (required by Amiga calls) within the function.

Each function requires a 'file-pointer', which will have been acquired via FOPEN , 0FOPEN or $FOPEN . File pointers are not considered addresses and are used just as AmigaDOS returns them; they are NEVER converted to relative.

Their names and stack diagrams are as follows:

FEMIT (file-pointer char -- , emits character to file)

This will abort if an error occurs.

FKEY (file-pointer -- char , gets character from file)

This will abort if an error occurs. It is not recommended that this be used in commercial aplications because it does not handles gracefully. But it is handy.

FREAD (file-pointer addr cnt -- #read | -1)

The FREAD and FWRITE functions are straightforward in their operation, operating on the specified memory and file (at its current address). Each return the number of bytes processed, or -1 if an error occurred.

FWRITE (file-pointer addr cnt -- #written | -1)

FSEEK (file-pointer filepos mode -- prevpos | -1)

The FSEEK 'mode' parameter equates directly to the AmigaDOS declared parameters OFFSET_BEGINNING, OFFSET_CURRENT, and OFFSET_END. For example, to seek to the end-of-file minus 5 bytes:

```
MYFILE @  -5  OFFSET_END  FSEEK .
```

These 5 calls will also set a user variable, FERROR, if appropriate. (If you want to check FERROR, do so immediately after the function returns. It will be reset by the next file operation.)

Following are various examples of reading from and writing to a file after it has been opened and the file-pointer stored in a VARIABLE called MYFILE.

Change the current location to the beginning of the file:

```
MYFILE @  0  OFFSET_BEGINNING FSEEK  ( -- ret-code )
```

Read 100 bytes from current position, place them at PAD:

```
MYFILE @  PAD  100 FREAD   ( -- ret-code )
```

Write 100 bytes from PAD to the file at its current location:

```
MYFILE @  PAD  100 FWRITE  ( -- ret-code )
```

Closing Files.

The normal method of closing a file opened under JForth is:

FCLOSE (file-pointer -- , return file resources to AmigaDOS)

A normal program then, will use FOPEN when it starts, and FCLOSE at its completion. In development environments, however, applications often will not finish as an error condition may cause it to QUIT. In JForth, you may optionally mark your file to be automatically closed in this event by executing MARKFCLOSE on a duplicate of the just-opened file-pointer. If your application successfully completes, you should UNMARKFCLOSE your file(s), so that they are not closed multiple times.

MARKFCLOSE (file-pointer -- , mark file to auto-close at quit)

UNMARKFCLOSE (file-pointer -- , remove from 'auto-close' stack)

Example: illustrates FOPEN, MARK and UNMARKFCLOSE and FCLOSE

```
: EXAMPLE   ( -- , parses a filename from the input stream )
  FOPEN  ( -- file OR false )  ?dup
  IF  dup MARKFCLOSE   ( file -- )
```

```
                \ auto-close it at QUIT.
        MYFILE !          ( -- )
                \ save it in my variable
        Do-My-Thing       ( -- )
               \ do file processing, whatever it is
        MYFILE @          ( file -- )
               \ fetch the file pointer
        dup UNMARKFCLOSE ( file -- )
               \ remove from the auto-close stack
        FCLOSE            ( -- )
               \ and close it!
      ELSE  cr ." File could not be opened!"  QUIT
      THEN
;
```

Building AmigaDOS Filenames.

Three words help you build null-terminated strings for AmigaDOS, but are not normally needed; FOPEN can usually be used to specify a file. These are handy to modify filenames algorithmically and resubmit them to AmigaDOS (via the 0FOPEN word). They are:

```
DOS0  ( -- addr , returns the address of the NUL-string buffer )
>DOS  ( addr cnt -- , place string in DOS0, NUL terminated)
+DOS  ( addr cnt -- , append this string to one already at DOS0)
```

Note that >DOS and +DOS maintain a count byte usable by the JForth string words. For example, the contents of DOS0 can be typed by:

```
DOS0 1- count type
```

Here is an example of using these words to build a file pathname.

```
: FOPEN.DATA  ( <name> -- , append suffix and open )
    FILEWORD COUNT >DOS
    " .DATA" COUNT +DOS
    DOS0  0FOPEN
;
FOPEN.DATA  EXPT1  ( open "EXPT1.DATA" )
```

Sequential Virtual File Utilities

Several words provided allow easy use of a 1024 byte virtual buffer area for file words designed to sequential single-character or cell-based transfers. Using these words, a program may realize a significant speed improvement for certain types of file I/O.

An application, at its start, may open a file-virtual buffer and store the resultant address in a variable. The buffer may be used by passing the address of the variable (not the buffer) as an argument to certain virtual-calls.

Also, some JForth-provided functions (such as READLINE) are only accessible through this scheme.

Those words dealing with virtual buffer management include:

OPENFV (var-addr -- buffer-addr)

Allocate a 1K buffer, and set the variable to its address. Even though the buffer address is not normally needed by applications, it is returned. If a zero is returned, an error occured.

CLOSEFVREAD (var-addr --)

Deallocate the buffer being pointed to by VAR. This buffer has only been read from. Clears the variable.

CLOSEFVWRITE (file-pointer var-addr --)

Flush any leftover data to the file, deallocate the buffer and clear the variable. (This buffer may have only been used for writing).

F, (file var n1 --)

Send 'n1' (32 bits) to the next available cell in FILE via the virtual buffer stored in VAR.

READLINE (file var addr-addr maxcnt -- addr cnt | addr -1)

Read FILE via the buffer stored in VAR, place at ADDR, do not exceed MAXCNT characters. Returns 0 if an empty line, -1 if end-of-file.

TEMPF, (n1 --)

Same as 'F,' but uses TEMPFILE and TEMPBUFF.

As an example, observe this simple word which opens the file whose name follows in the input stream, then types each line to the screen until the end of file. (Note: TEMPFILE & TEMPBUFF are user variables that are pre-defined in JForth for such uses.)

```
: LISTFILE   ( -- , eats name )
\ types inputted FILENAME to console
   FOPEN -dup
   IF   TEMPFILE !    \ save file pointer
\ allocate virtual buffer
      TEMPBUFF OPENFV ( addr -- )
      drop
\ tempbuff inited by OpenFV, don't need addr
      BEGIN
         TEMPFILE @  TEMPBUFF
         HERE 1000   READLINE
         DUP 0 < 0=
         ( addr #read true-if-not-eof -- )
      WHILE  CR TYPE
      REPEAT 2DROP
      TEMPBUFF CLOSEFVREAD   \ deallocate the buffer
      TEMPFILE @ FCLOSE           \
   ELSE cr ." Can't open "  DOS0 1- count type  quit
   THEN cr
;
```

DOLINES - Easy Text File Processing

The DOLINES system provides a simple way to process text files on a line by line basis. You can set a deferred word that will get called for each line of the file. It will be passed the line as a string. You can then do whatever you want with that string. Here is an example of a program that types a file to the screen.

First define a word that will process each line as it is read.

```
INCLUDE? DOLINES JU:DOLINES

: SHOWLINE ( $line -- , type it with line number )
    CR DL-LINENUM @ 5 .R
    SPACE  $TYPE ?PAUSE
;
```

DL-LINENUM is the line number that is maintained by DOLINES. Now write a word that will set the vector and call DOLINES.

```
: SHOWFILE ( <filename> -- , print file to screen )
    ' SHOWLINE IS DOLINE ( set deferred word )
    DOLINES
;
SHOWFILE JU:BSORT
SHOWFILE JU:ANSI
```

Once the vector DOLINE is set, you can call DOLINES which will take a filename from the input stream, open the file, and pass each line to DOLINE.

$DOLINES ($filename --)

Same as DOLINES but takes name on stack as string.

DL-LINENUM (-- addr , variable containing current line number)

DL.CLOSE.FILE (-- , close the doline file)

You should call this from the word that you set DOLINE.ERROR to.

DOLINE ($LINE -- , do something!?!)

This is a deferred word that the user can set to anything they want as long as it has the same stack diagram.

DOLINES (<filename> -- , open and process file)

DOLINE.ERROR (--)

Deferred word that is called if an error is encountered while processing the file. See DL.CLOSE.FILE.

Chapter 9
Floating Point Arithmetic

Dave Sirag has graciously donated his implementation of the Forth Vendors Group Standard for use with JForth. This standard supports numerous arithmetic and comparison operators and also provides very flexible numeric conversion tools. Dave has also added numerous extensions that we think you will find helpful. We at Delta Research are very grateful to Dave for his contribution.

There are two main files for the floating point code, JFLT:FLOAT.FFP and JFLT:FLOAT.DOUBLE. The first file supports 32 bit single precision floating point using the Motorola Fast Floating Point Library. The second file supports 64 bit double precision IEEF format floating point for when extreme accuracy is more important than speed. The command sets between the two are essentially identical making it possible to write utilities that will work with either precision.

Without further ado, let's load the 32 bit code and try it out.

Floating Point Tutorial

First we must load the appropriate Floating Point package. I suggest starting with the single precision. If you are asked for the workbench disk please insert it as requested.

```
INCLUDE? F* JFLT:FLOAT.FFP
```

If you see a message about words not being compiled INLINE, do not worry. This just means that they could notbe compiled in the fastest possible form. It is not harmful. See INLINE in the glossary.

Now we *must* initialize this system before using it. Enter:

```
FPINIT
```

That will open up the appropriate libraries and install the proper floating point number conversion routines. Now whenever we enter a number with a decimal point in it, it will be automatically converted to floating point.

Simple Arithmetic and Output

Let's try entering some numbers and doing some simple arithmetic. Most of the arithmetic operators that Forth has for integers, "* + - / ABS MIN", etc. have their floating point counterparts. To add two floating point numbers together we simply call F+ .

```
23.5 F.  ( print a fp number )
20.0 7.55 F+ F.  ( add two fp numbers )
17.98 12.345 F/ F.  ( divide two fp numbers )
10.0 PI F* F.
```

We can also enter and display numbers in exponential notation.

```
23.7E15 E.
7.193E-8 ENG.  ( engineering format )
```

Engineering format always adjusts the exponent to a multiple of three. This is convenient for displaying seconds, milliseconds, and microseconds, or meters and kilometers, etc.

We can control the display of our numbers by using the ".R" words. If we want to display PI with 4 places after the decimal point in a field 12 characters wide, we can enter:

```
PI 4 12 F.R
8.98 99876.3 F/  2 10 ENG.R
```

Transcendental Functions

Scientific calculations often require transcendental functions like sine and cosine. This package assumes angles to be in Radians instead of degrees. Radians are a special unit of angular measurement where 360 degrees (a circle) is two Pi, or roughly 6.3, radians. Let's do some calculations. (Don't worry! If you don't understand this stuff then you probably don't need it.)

```
PI  FSIN F.  ( should be zero )
45.0 DEG>RAD FCOS F.  ( approx 0.7 )
```

Notice the use of DEG>RAD to convert between degrees and radians. RAD>DEG can be used to go the other way.

```
2.1  5.3 F** F.  ( raise 2.1 to the power of 5.3 )
137.2 FLOG F.    ( log base 10 )
13.72 FLOG F.
```

Let's calculate the hypotenuse of a triangle. Pythagoras' theorem tells us that the length of the hypotenuse of a right triangle is the square root of the sum of the squares of the two sides.

```
: HYPOT  ( a b -- c , C = SQRT( A**2 + B**2 )
    FDUP F*  ( square B )
    FSWAP FDUP F* F+ ( square A and add )
    FSQRT  ( take square root )
;
3.0 4.0 HYPOT F.  ( yep, the old 3,4,5 right triangle)
9.8 17.5 HYPOT F.
```

Precision Independent Style

Suppose you write an application that works for 32 bit single precision numbers. Wouldn't it be nice if it also worked for double precision without having to change it. By carefully using the special stack operators and storage words that Dave has written, you can do just that. The tricky part is that 32 bit numbers occupy one cell on the stack and 64 bit numbers occupy two cells. If you have two 32 bit numbers on the stack and you want to swap them you can just use SWAP . If you have two 64 bit numbers you have to use 2SWAP . If you have mixed integer and floating point things get even trickier. Luckily most of the common stack operations have been provided in a form that works no matter what precision we are using. Enter:

```
43  29.712 NFSWAP . F.
```

This version of SWAP assumes that the stack has an integer, N, and a float, F. It is defined differently for different width floating point numbers. For single precision it is just a normal SWAP , for double precision it uses ROT to do its work.

We also need special words for VARIABLE , ! and other width-dependent words. Enter:

```
FVARIABLE FVAR-1
23.4 FVAR-1 F!
FVAR-1 F@ F.
```

That will work for single or double precision. Now, suppose you want an array of numbers.

```
20 FARRAY MY-FAR  ( declare array of 20 numbers )
( store 9.876 into fourth location in array )
9.876 4 MY-FAR F!
```

Cloning Floating Point Code

Please remember that you must initialize the system before using it. Otherwise it will crash dramatically. Here is an example of a floating point program that will clone.

```
: SHOWSINES  ( -- , display sines )
    FPINIT  ( Important!! ) CR
    91 0
    DO  I DUP 4 .R  ( show angle )
        FLOAT DEG>RAD ( convert i to radians )
        FSIN   4 9 F.R CR
    LOOP
    FPTERM
;
```

Floating Point Glossary

This glossary has been organized by function to make it easier to find the right word.

Floating Point Control

FPINIT (--- , initialize floating point)

This *must* be called before using any floating point words or you will crash. It opens the appropriate libraries and initializes some variables. The floating point files have AUTO.INIT words that will automatically call FPINIT if they are loaded permanently in a dictionary.

FPTERM (--- , close libraries and cleanup)

Arithmetic Operators

These operators are similar to the corresponding integer operators and, therefore, don't need much explanation.

```
F+    ( r1 r2 --- r1+r2 )
F-    ( r1 r2 --- r1-r2 )
F*    ( r1 r2 --- r1*r2 )
F/    ( r1 r2 --- r1/r2 )
F2*   ( r --- r*2.0 )
F2/   ( r --- r/2,0 )
FABS  ( r --- |r| , take absolute value of R )
FNEGATE ( r --- -r )
FMAX  ( r1 r2 --- r1 | r2 , leave largest of r1 and r2 )
FMIN  ( r1 r2 --- r1 | r2 , leave smallest of r1 and r2 )
```

Result Flags

The arithmetic operators above set a variable called FPSTAT to the value in the condition code registers after the operation. These words examine FPSTAT and leave a flag if a given condition is true. They can be used to determine if an overflow had occurred or if the result was zero, etc. An overflow is what a happens when a calculation results in a number that is too big or too small for the machine.

```
FEQ  ( -- flag , true if result zero )
FLE  ( -- flag , true if result less than or equal to zero )
FLT  ( -- flag , true if result less than zero )
FGE  ( -- flag , true if result greater than or equal to zero )
FGT  ( -- flag , true if result greater than zero )
FNE  ( -- flag , true if result not equal zero )
FVC  ( -- flag , true if result did NOT overflowed )
FVS  ( -- flag , true if result overflowed )
```

Transcendental Functions

Angles are passed in radians. Use DEG>RAD or RAD>DEG to convert if needed.

```
DEG>RAD    ( r.deg -- r.rad , convert degrees to radians )
DEG/RAD    ( --- r , constant number of degrees per radian )
F**        ( r1 r2 --- r1**r2 , R1 to R2th power )
FSQRT      ( r --- r**0.5 , take square root )
FLN        ( r --- ln[r] , natural logarithm )
FLOG ( r --- log[r] , base 10 logarithm )
FALOG      ( r --- 10**r , inverse logarithm )
FALN ( r --- e**r , inverse natural log )
FSIN ( r.rad --- sin[r] , take sine of R )
FCOS ( r.rad --- cos[r] )
FTAN ( r.rad --- tan[r] )
FASIN      ( sin[r] --- r.rad , take arcsine of R )
FACOS      ( cos[r] --- r.rad )
FATAN      ( tan[r] --- r.rad )
FSINH      ( r.rad --- sinh[r] , take hypersine of R )
FCOSH      ( r.rad --- cosh[r] )
FTANH      ( r.rad --- tanh[r] )
FCS        ( r.rad --- sin[r] cos[r] , calc both quickly )
FATCS      ( sin[r] cos[r]--- r.rad , four quadrant arctangent )
```

This is similar to the ATAN2 function in FORTRAN

```
PI         ( -- pi , constant )
PI/2 ( -- pi/2 , constant )
2PI        ( -- 2pi , constant )
RAD>DEG    ( r.rad -- r.deg , convert radians to degrees )
```

Logical Operators

These operators are just like their integer counterparts except they accept floating point numbers. The following words accept two numbers.

F= F< F> F<= F>= F<> (r1 r2 -- flag)

The following words compare a number to zero.

F0= F0< F0> F0<> F0<= F0>= (r -- flag)

Stack Operators

Since floating point numbers may be 32 bits or 64 bits, we need a way to manipulate them on the stack without knowing their size. Then we can write programs that will work with either precision.

```
F>R         ( r --- , push to return stack )
FCELL+      ( n --- n' , add width of floating point number )
```

This word will add the width of a floating point cell. For single precision this would be 4+. For double precision would be 8 +.

```
FCELL-      ( n --- n' )
FCELLS      ( n --- n' )
FCELL/      ( n --- n' )
FCELLU/     ( n --- n' )
FDUP ( r -- r r )
FNOVER      ( r n --- r n r )
FNSWAP      ( r n --- n r )
FFNROT      ( r1 r2 n --- r2 n r1 )
FNFROT      ( r1 n r2 --- n r2 r1 )
FNNROT      ( r n1 n2 --- n1 n2 r )
FOVER       ( r1 r2 -- r1 r2 r1 )
FR>         ( --- r , pop from return stack )
FROT ( r1 r2 r3 --- r2 r3 r1 )
FSWAP       ( r1 r2 -- r2 r1 )
NFOVER      ( n r --- n r n )
NFFROT      ( n r1 r2 --- r1 r2 n )
NFNROT      ( n1 r n2 --- r n2 n1 )
NFSWAP      ( n r --- r n )
NNFROT      ( n1 n2 r --- n2 r n1 )
```

Number Storage

These words provide a precision-independent way of storing floating point numbers in memory and retrieving them.

```
F!          ( r addr --- , store in memory )
F@          ( addr --- r )
FARRAY      ( n <name> --- , declare an array with N cells )
FCONSTANT ( value <name> --- , declare a big enough constant )
FVARIABLE   ( <name> --- , declare a big enough variable )
```

Number Conversion Operators

For the following set of words, N is 32 bit for both double and single precision.

```
FIX         ( r --- n , round and convert to integer )
FLOAT       ( n --- r , convert integer to float )
INT         ( r --- n , truncate and convert to integer )
PACK ( d n --- r )
UNPACK      ( r --- d n )
```

Display Operators

The single precision display words support 7 significant figures. Thus 1.2345678 F. will display 1.234568 .

```
E.   ( r --- , display floating-point in exponential form )
E.R  ( r places width --- , exp form display of  R )
ENG. ( r --- , display R in engineering exponential form )
ENG.R       ( r places width --- , formatted engineering display )
F>ENGTEXT   ( r --- addr count , converts fp to eng-form text )
F>ETEXT     ( r --- addr count , converts fp to e-form text )
F>TEXT      ( r --- addr count , converts fp to text )
F.          ( r --- , display floating-point in decimal form )
F.R         ( r places width -- , formatted decimal display of R)
PLACES      ( n --- , sets default number of fractional digits )
```

Display Operators & Variables

These variables can be used to control the display of numbers. They are extensions to the FVG84 standard.

DP-CHARS (--- addr , symbols for decimal point and comma)

Allows other styles of display. The characters in this are packed as two 16 bit numbers. Here is an example of changing the decimal point and the commas.

```
COMMAS
2.345,67 F.  ( normal )
ASCII ^ DP-CHARS W!
ASCII - DP-CHARS 2+ W!
2.345,67 F.
```

E.PLUS (--- addr , if true, display "E+01" rather than "E01")

EFFLD (--- addr , width of fractional field for E. and ENG)

EXPSYMBOL (--- addr , char for "E" for E. and ENG.)

F#BYTES (-- #bytes , per float = 4 for single, 8 for double)

F.ENDPOINT (--- addr , if true, use a point at end of F.)

FFLD (--- addr , - width of fractional field for F.)

This determines the number of places after the decimal point. Range is 0 to 7 for single precision.

FLD (--- addr , total width of field for number displays)

F.EXMAX (--- addr , maximum exponent for decimal F.)

If the exponent is larger than this it will use the E. format.

FPWARN (--- addr , if true, display fp warning messages)

Number Interpreters

Floating point numbers will always be converted using base 10 (decimal) regardless of the value of BASE. This means no HEXADECIMAL floating point numbers. These determine how numbers will be input.

FLOAT.INTERPRET (--- ,allows integer decimal, and "E" form input)

FASTFP.INTERPRET (--- , allows integer and decimal form input)

FIX.INTERPRET (--- , integer input only, '.' implies double)

FNUMBER? ($string -- r true | false)

Converts a string to a floating point number and true if valid. Otherwise returns false.

NTYPE (--- addr , type of last number converted)

This will be set when a number is input. The values are 1 = int, 2 = fp, 0 = not number.

Chapter 10
Object-Oriented Development Environment (ODE)

Note: ODE was developed by Phil Burk to support HMSL, the Hierarchical Music Specification Language. ODE was then released as a part of JForth in an effort to promote object-oriented programming. This chapter, therefore, appears in both the JForth manual, and the HMSL manual. When information specific to JForth or HMSL appears, it will be noted as such.

Philosophy

Object-Oriented Programming (OOP) allows you to design programs in a way that more closely matches the real world. In the world, we are surrounded by objects. These objects can be thought of as belonging to different *classes*, for example pens and pencils. These can in turn be thought as belonging to larger, more general classes such as writing implements. In the same way, an object-oriented program involves software based *objects*. If you were writing a program for an airline you would have classes such as airplanes, airports, passengers, etc. Airplanes would have information associated with them such as flight numbers, fuel capacity, altitude, and so on. If you wanted a particular plane to climb to a new altitude, you could send it a message telling it to climb. It would then update its internal record of its altitude. For programs that are not tied so closely to the physical world, you might define classes of files, tables, arrays, or plots.

Quite often, the class of object that you need is already defined. Then you can just use it without having to define new ones. If you need something very similar to an existing class but different, you can define a new class that *inherits* the desired qualities of the existing class. The fact that classes can often be used in more than one program allows you to build up a library of classes. This can save time when programming.

A class defines what an object is made of and what it can do. Once you have defined a class, you can create as many instances (objects) of that class as you want. Objects may be thought of as intelligent data structures. Each object knows how to manipulate its own internal data. All objects of a given class will have the same internal structure and will use the same methods for manipulating that data.

Existing Classes in ODE

A few classes have been defined as part of the official ODE package. The most commonly used ones are:

```
OB.BARRAY  = byte array
OB.ARRAY   = long word array
OB.ELMNTS  = list of N dimensional points, can also be thought of
as a 2D array.
OB.LIST    = list of single values
OB.OBJLIST = list of other objects.
```

Hidden Data

The organization of the data and the actual techniques used to manipulate it are *hidden* from users of the object. It effectively isolates the information needed to manipulate an object inside the object itself. This promotes a very modular structure. The code that uses an object doesn't have to know anything about how that object was implemented. This makes it very easy to modify how an object

works without having to modify the code that uses it. This can be a great advantage when managing large software projects.

Generic Messages

When you want an object to perform some action, you send it a message. Examples of messages might be PRINT: or CLEAR:. Objects "know" how to respond to messages based on methods defined for their class.

An advantage of object-oriented programming is that you can send the same message to objects of different classes. As an example, you could send a PRINT: message to several different objects. Each one would know how to print its contents in a meaningful form. Some would print a table of values. Some would only print a single value. One advantage of this is that you don't have to memorize a differently named PRINT: function for each type of data structure. In a traditional procedural system you would be writing words like PRINT.ARRAY and PRINT.THING for different data structures.

Another advantage is that you can write generic code. If you have a picture containing different kinds of graphics objects, you don't have to know what they are or how to draw them. Just send each one a DRAW: message and they will each know how to draw themselves.

Tradeoffs

The use of object-oriented programming techniques can simplify software design, speed code development, reduce many types of common errors, and improve maintainability. When you use it for awhile, you will see why so many programmers are using these techniques in ODE and other OOP languages like C++ and Smalltalk.

I know what you're thinking now. There must be a catch. Well, one disadvantage with objects is that there is some extra memory and speed overhead. The object-oriented support code does use some dictionary space. It can also be slightly slower than traditional code. Standard Smalltalk is quite slow because everything is an object, including every variable, constant, etc. ODE, however, was designed for real time music applications and was, therefore, optimized for speed. This approach is too restrictive for a Forth implementation. ODE only makes objects out of the more complex data structures. All ODE objects use common code that has been optimized for speed.

OOP techniques can also seem strange and confusing at first. After using it for awhile, however, you will feel a small "click" in your head and it will all become clear. You may even find, like I do, that it is hard to imagine writing some programs without using OOP.

All things considered, OOP techniques can be a real advantage when developing software. I think you will find it useful and enjoyable.

Origins of OOP

The inspiration for ODE came from Smalltalk, the original object-oriented programming language developed at the Xerox Palo Alto Research Center. In an effort to promote standardization, the syntax of ODE is similar to the syntax for both Smalltalk and NEON, an object-oriented dialect of Forth for the Macintosh.

For more information about object-oriented languages and their characteristics, see the August 1986 issue of *BYTE* magazine. There is also an excellent textbook available called *Smalltalk-80: The Language and Its Implementation* by Goldberg and Robson. Another good text is *Object Oriented*

Programming by Brad J. Cox. These references and many others are in the bibliography at the end of this manual.

Terminology

Class - A description of the data contained within an object, and the methods are used to manipulate that data.

Object - An instance of a class. Each object has its own data space and a pointer to its class.

Method - A function that is associated with a particular class. By convention, method names generally end in a colon, for example, PRINT: or ADD:.

Superclass - The class from which a new class is derived. All classes are derived ultimately from a root class named OBJECT .

Inheritance - Each new class automatically has available all of the properties of its superclass. This includes all instance variables and all methods. The new class then adds new methods and/or instance variables.

Instance Variable - A data item that is contained within each object of a given class. To access an instance variable from outside an object, you must use only that object's defined methods. Instance variables can be hidden from other code by not providing any methods for accessing them. Instance variables can themselves be objects.

Instantiate - To create an object from its class definition.

Turorial 1 - Creating and Using Objects

Including ODE

ODE is already compiled as part of **HMSL. JForth** users can load ODE by entering:

```
INCLUDE? OBJECT JO:LOAD_ODE
```

Creating an Object, *Instantiation*

Creating an object is similar to creating other Forth data structures, like VARIABLE. Enter the class name followed by the name of the object to be defined. Let's create an instance of the integer class OB.INT named MY-INT. Enter:

```
OB.INT MY-INT
```

Please note that this since this is a defining word, it should be used outside of colon definitions, just like VARIABLE and CONSTANT .

You have now defined an object that exists in the Forth dictionary. If you enter the name of an object, it will return its relative address. You can use this address if you want to pass the object as an item on the stack. Enter:

```
MY-INT  .
```

Sending Messages

To make an object do something, you must send it a message. For example, to put a value into this integer object, send a PUT: message to it. The integer will know what to do. In this case it will know

to take the value at the top of the stack and store it internally.. You can then send a PRINT: message to verify that this worked. Here is an example:

```
456 PUT: MY-INT
PRINT: MY-INT ( 456 will get printed. )
```

The OB.INT class is not very useful except as an example. It was written as a simple exercise and is often used as a superclass for other classes. When I want an INTEGER I usually use a VARIABLE or a VALUE.

Using Arrays

A more useful class of objects is the array. To create an array of 32-bit words named MY-ARRAY, enter:

```
OB.ARRAY MY-ARRAY
```

The data for the array is stored in dynamically allocated memory that we request from the operating system. This allows small programs to access large amounts of memory whose size can vary as needed. To allocate memory, send a ?NEW: message to the array. The array object will request that enough space for that much data be allocated from a pool of free memory. To allocate room for 10 cells, enter:

```
10 ?NEW: MY-ARRAY .
```

Since each array cell is four bytes, 10*4=40 bytes were just allocated. If the allocation was successful, the address of the allocated memory is returned. If the allocation fails because of insufficient free memory, it will return a zero. You should always check to make sure that you got the memory you requested. No matter how much memory you have installed, you can always run out of free memory. If you are writing programs for others, remember that they may not have as much memory as you do.

Now let's see what is in our array.

```
PRINT: MY-ARRAY
```

When you printed that object, it was probably full of strange numbers. Those are the values that just happened to be in the memory you allocated. To clear it, enter:

```
CLEAR: MY-ARRAY
PRINT: MY-ARRAY
```

Notice that you do not have to use different messages for printing integers and arrays. This means that there will be fewer commands to memorize. You might try sending a CLEAR: message to your integer to see what happens.

To access individual items inside your array, you can use the messages AT: and TO: . For example:

```
98 4 TO: MY-ARRAY ( Store a 98 at item number 4)
101 3 TO: MY-ARRAY
0 AT: MY-ARRAY . ( Fetch and print value in cell 0, the 1st cell )
PRINT: MY-ARRAY
```

Finding an item in an Array

So far this looks like a pretty standard array. (See ARRAY in the main glossary for an example of a standard array.) Because it is an object, however, it can be smarter than a standard array. Let's ask the array to find a value inside of it. Assuming you enterred the examples above, let's ask it to find the first occurence of 101. Enter:

```
101 INDEXOF:  MY-ARRAY .S
```

Notice that it returned a 3 and a -1. The 3 is the index of the value 101 and the -1 is a TRUE value. If it can't find the value it just returns FALSE.

```
0SP   1969  INDEXOF: MY-ARRAY .S
```

Range Checking

A common error that can occur when programming is to use an array index that is too large for the array. Our array has 10 items in it. The indices run from 0 to 9. Let's try to read past the end of our array.

```
10 AT: MY-ARRAY
123 900 TO: MY-ARRAY
```

ODE can check for indices that are out of range and will abort if it detects one. Notice that it printed several things here. The index out of range is printed first. Then ODE dumps the object stack. (More about this later.) The current object is the bottom one listed. Then it prints the nature of the error. This is especially handy when debugging. Once you have an application thoroughly debugged, you can turn off this range checking. One way to do this is to enter:

```
FALSE DO.RANGE: MY-ARRAY
10 AT: MY-ARRAY  \ you get a number but it is garbage
```

Warning, do **not** try to see if TO: will let you over-index. AT: is safe but over-indexing TO: can overwrite memory and cause you to crash.

Another way to turn off range checking is to enter RUN.FASTER and recompile your program. This will affect all objects compiled.

If you forget how many items you have, you can use LIMIT:. Enter:

```
LIMIT: MY-ARRAY .
```

Freeing Memory in Array Classes

Many classes use the NEW: method to allocate memory for their use, as we have seen. When one is finished using an object, one must DEallocate the memory. This is done with the FREE: method.

```
FREE: MY-ARRAY
```

I recommend writing a word that frees all the objects that need to be freed. You can then use IF.FORGOTTEN to make sure this word is called automatically if you forget the code that defines them. Once you FORGET an object, it is too late to FREE: it..

```
: CLEANUP
   FREE: MY-ARRAY
;
IF.FORGOTTEN IF.FORGOTTEN
```

}STUFF: and FILL:

Let's now try to stuff some specific numbers into our array. One easy way to do this is using }STUFF:. Enter:

```
STUFF{  23  987  44  2001 }STUFF: MY-ARRAY
PRINT: MY-ARRAY
```

Notice that }STUFF: automatically allocates memory if needed. If you want you can call ?NEW: before }STUFF: to guarantee allocation.

To fill an entire array at once, enter:

```
345 FILL: MY-ARRAY
PRINT: MY-ARRAY
```

Now that we are done, enter:

```
FREE: MY-ARRAY
```

Tutorial 2 - Early versus Late Binding

To Whom It May Concern,

In the previous tutorial, we sent messages to a specific object, MY-ARRAY. This would be like sending a message to a specific person. We would put that person's name on the message, "Dear Larry, blah blah blah". Sometimes, however, we don't have any specific person in mind to receive the message. If a shopkeeper goes to lunch, he or she would put a "Gone to Lunch" message on their door. The message would then be received by anyone who happened to stand in front of that door. Similarly, we can send a message to *whatever object happens to have its address on the top of the stack*. When we write the message we may not know what object that is so we cannot use its name in the message.

The word **[]** (a left square bracket followed by a right square bracket pronounced "bracket bracket") can be used to send a message to an object on the stack. ODE programmers also pronounce this as “late-bind.” The following two lines are essentially equivalent.

```
PRINT: MY-ARRAY
MY-ARRAY PRINT: []  ( MY-ARRAY leaves its address on stack)
```

The first technique is called *early binding* and the second is called *late binding*. When a word is compiled with early binding, the CFA (Code Field Address) of the method to use is determined at compile time and compiled into the dictionary. For late binding, this process doesn't happen until run time. Late binding is therefore slightly slower than early binding but is often required for its added functionality.

This may sound more complicated than it really is in actual use. Late-binding is simply a technique for sending a message to an object addressed “to whom it may concern.” The main difference is that in late-binding, the object’s address comes *before* the method, and the method is followed by the late-bind brackets.

Imagine that you wanted to define a word that would print then clear the contents of many different objects. This could easily be done with late binding. (You might want to put this next example in a file because we will change it later.)

```
: PRINT&CLEAR ( object-address -- , print then clear an object )
   DUP   ( duplicates the object address )
   PRINT: [] ( use late-binding )
   CLEAR: []
;
MY-INT PRINT&CLEAR ( pass the address of MY-INT )
MY-ARRAY PRINT&CLEAR
```

Local Variables and Late Binding

A useful technique with late-binding is to use local variables to hold the address of the object. (Please familiarize yourself with local variables before continuing. JForth users will find them in chapter 11. Macintosh users will find them described in the Macintosh supplement.) Locals can help eliminate confusing stack manipulations. As an example, you could slightly modify the above word to store the address of the object in a local variable:

```
: PRINT&CLEAR { obj -- , print then clear an object }
   OBJ PRINT: []
   OBJ CLEAR: []
;
```

Since this technique is used so often with local variables, ODE supports an alternative syntax. Local variables that contain an object address can be used as if they were an object. *The message is still late bound*, but it is easier to read. Here is another way to write the above word.

```
: PRINT&CLEAR { obj -- , print then clear an object }
   PRINT: OBJ
   CLEAR: OBJ
;
```

This example is rather trivial. The power of using these local variables will be more apparent when used in more complex words.

Now that we are done, don't forget to enter:

```
FREE: MY-ARRAY
```

Tutorial 3 - Using OB.ELMNTS

A very useful subclass of OB.ARRAY is OB.ELMNTS. Each *element* of this array can have multiple values. An example would be an array of X,Y points, or elements. Each element would have 2 *dimensions*, X and Y. In a 3 dimensional array, each element would have an X, a Y and a Z value. The space need not be geometric. Another 3 dimensional space could have the dimensions Time, Pitch, and Loudness. The elements in this space could be musical notes. Let's look at an example of an array of X,Y points. Let's make room for 100 points with 2 dimensions. Enter:

```
OB.ELMNTS  XYPS  \ x,y points
100 2 ?NEW: XYPS .  \ if zero then not enough memory!
```

When we printed the array, we saw that there were no points in the array yet. Let's add some. Enter:

```
10 52 ADD: XYPS
30 17 ADD: XYPS
15 294 ADD: XYPS
PRINT: XYPS
```

ADD: is based on the notion that elements have *no* value until a value has been given them. In reality, of course, every byte in a computer has a value from the moment it is turned on. In this model, however, even though data memory has been allocated, the object is considered to be initially empty. You can add elements one at a time by using ADD: and have it keep track of how many you have added. You can find out how many elements have been added by using MANY: .For example:

```
MANY: XYPS . ( Prints '3' )
```

We can access the elements in the array using GET: and PUT:. These take an index and operate on the whole element. Let's change the value of the second point. Remember that elements are numbered starting with zero so the second point is number 1. The first point is number 0.

```
50 99 1 PUT: XYPS
PRINT: XYPS
1 GET: XYPS .S
CR SWAP . .
```

If you want to access an individual number, you can use ED.TO: and ED.AT:. The "ED" stands for **e**lement and **d**imension to help you remember how to pass the indices. To change the value in element 2, dimension 1, enter:

```
75 2 1 ED.TO: XYPS
PRINT: XYPS
2 1 ED.AT: XYPS .
```

We can ADD: more elements to the end or we can insert elements anywhere in the middle or at the beginning. Enter:

```
63 444 1 INSERT: XYPS \ before element 1
PRINT: XYPS
```

We can remove an element as well. Enter:

```
0 REMOVE: XYPS
PRINT: XYPS
```

Another way to access the elements is sequentially. We can get the first element using: FIRST: then continue using NEXT:. Enter:

```
FIRST: XYPS SWAP . .
NEXT: XYPS SWAP . .
MANYLEFT: XYPS .  \ just one left to process
NEXT: XYPS SWAP . . \ that's the last one
```

If we go to far with NEXT we will get an error. Enter:

```
NEXT: XYPS
```

We went past the end and got an error. If we want to keep going forever we could use NEXTWRAP:. It will wrap around back to the first element when it reaches the end.

The *read pointer* can be set to a specific point or reset back to the beginning.

```
1 GOTO: XYPS  \ set to read #1
NEXT: XYPS SWAP . .
WHERE: XYPS .  \ where are we now?
RESET: XYPS
NEXT: XYPS SWAP . .
```

If you don't want to ADD: points, you can use SET.MANY: to make it seem as if there are many points. Enter:

```
50 SET.MANY: XYPS
PRINT: XYPS
77 0 FILL.DIM: XYPS
PRINT: XYPS
```

When you want to get rid of those points, use EMPTY: which sets many to zero. (Don' t forget that you can save typing by using <UP-ARROW> to reenter PRINT: XYPS)

```
EMPTY: XYPS
PRINT: XYPS
50 SET.MANY: XYPS
PRINT: XYPS
```

If you want to actually clear the data use CLEAR: as follows:

```
CLEAR: XYPS
PRINT: XYPS
50 SET.MANY: XYPS
PRINT: XYPS
```

When you are done using the array, PLEASE free the memory. Enter:

```
FREE: XYPS
```

Predefined Classes

A number of predefined classes already exist in ODE. They can be used directly, or as the superclasses for newly defined classes.

OBJECT

The class OBJECT is the root class for all other classes. You will probably never use it directly, however, because all of the methods described here will work for all other classes. This is because all classes *inherit* this class' methods.

ADDRESS: (-- ivars-address , address of instance variables)

.CLASS: (-- , print the class of an object)

DUMP: (-- , dump object's instance variables in hex)

GET.NAME: (-- $name , get printable name.)

This name can be used for printing graphically with GR.TEXT or for writing to a file.

NAME: (-- , print the name of the object)

This is usually used with late binding where the name is not known. NAME: SELF is often handy in error messages.

PUT.NAME: ($name -- , change the name of an object.)

The default name is the dictionary name. This name is only used for error reporting and printing. *If you change the name, you must still send messages using the name in the dictionary.* This is very important to remember, especially when using dynamically instantiated objects (see the discussion later in this chapter).

SPACE: (-- nbytes , return size of instance variable space)

This does not include specially allocated memory for arrays, etc. It does include the space required for the memory pointers, limits, etc. This method is not used very often.

OB.INT - subclass of OBJECT

This provides a simple integer object. Its superclass is OBJECT

CLEAR: (-- , clears instance variables)

GET: (-- value, returns value of integer)

PRINT: (-- , print value)

PUT: (value -- , sets value of integer)

+: (n -- , add n to value of integer)

OB.BARRAY - subclass of OBJECT

This is the basic array class. Its methods will also work for the other array classes. The data for the array is stored in memory allocated for it by the NEW: method. See "Using Arrays" above. The "elements" of the array are referred to as *items* to avoid confusion with the term *element* which has special meaning for OB.ELMNTS. The numbering of the items starts at zero. Thus an array with ten (10) items will have items numbered from zero (0) to nine (9).

+TO: (value index -- , add value to the indexed item)

?NEW: (#items -- addr | 0, allocate memory for the array)

This will automatically free any memory which has already been allocated, then allocate a new memory area. If memory cannot be allocated, a zero will be returned. If you run out of memory, quit from other programs or buy some more. See FREE:.

AT: (index -- value)

Return the value of an indexed item.

CLEAR: (-- , sets every item to 0)

This executes a FILL: with zero.

DATA.ADDR: (-- data-address)

Get address of allocated memory. This might be used on the Amiga version of HMSL, when using, for example, OB.BARRAY as an audio waveform.

DO.RANGE: (flag -- , enable or disable range checking)

EXTEND: (#items --)

Extend the memory allocated. This allocates a new area of memory and copies the old data to that new area. The old area is then deallocated. This is useful if you run out of items in an array. It is slow, however, so don't use it too often.

FILL: (value --)

Set every item in the entire array to value.

FREE: (-- , free the memory allocated by the NEW: method)

If you are finished using an array, use this method to deallocate its memory. If you do not deallocate memory for arrays in this way, the computer's memory will slowly get used up. It's good programming practice to FREE: all your objects when you're finished with them (like for example, when a piece is over).

INDEXOF: (value -- index true | false , search array for value)

LIMIT: (-- #items , return the number of items allocated)

This is used for setting DO … LOOP indices, checking for out-of-range conditions, etc.

NEW: (#items -- , allocate memory for the array)

This calls ?NEW: and aborts if it returns zero.

RANGE: (index -- , range check index)

This will abort with an error message if the index exceeds the limit of the memory allocated. This is performed automatically but can be disabled using DO.RANGE: .

SET.WIDTH: (#bytes -- , Set width in bytes for array item)

This allows you to have 1-, 2-, or 4-byte wide items in the array. The value must be set before any calls to NEW: , so that the right amount of memory can be allocated.

}STUFF: (v0 v1 v2 ... vN -- , puts values in array)

Use with STUFF{ to load an array. If needed, NEW: will be called to make room for the values.

```
STUFF{  12  34  987  6  }STUFF: MY-ARRAY
```

TO: (value index -- , set the indexed item to the value)

USE.DICT: (flag --)

If flag is TRUE, then ?NEW: will allot space in the dictionary when called instead of allocating dynamic memory. This allows you to initialize an array with data at compile time and save it using SAVE-FORTH. Only store numeric values in such an array, not addresses since they will not be valid at a later time.

WIDTH: (-- #bytes , width of a single array item)

Example of Using Arrays

Put this part in a file.

```
OB.BARRAY BAR1
: RAMPUP ( -- , fill with increasing even values )
   32 NEW: BAR1 ( allocate memory )
   LIMIT: BAR1 0
   DO
      I 2* ( even number )
      I TO: BAR1 ( store 2*I at Ith item )
   LOOP
;
```

INCLUDE the file then test it using by entering:

```
BAR1 RAMPUP ( put increasing values in array )
PRINT: BAR1
5 AT: BAR1 . ( will print 10 )
99 AT: BAR1 ( will report an "Index Out of Range" error )
FREE: BAR1
```

OB.ARRAY

This class is the same as the OB.BARRAY class except that each item is as wide as a standard stack item, which in JForth and HForth is *4 bytes*. Historically, this sensitivity to stack-width is important because HMSL has run on a wide variety of platforms and different versions of Forth (including 16- bit versions). The EXEC: method has been added to this class but the indexing is the same.

EXEC: (index -- , executes CFA stored at indexed item)

This assumes that you have stored some CFAs in the array to begin with. You may find it useful to fill such arrays with the CFA of an error handling routine. Then put in the specific CFAs you need. The following example shows how to get a properly handled error if you don't execute one of your specific routines:

Example of Execution Array

Put this part in a file.

```
: BAD.INDEX ( -- , report error )
   ." Invalid execution index." CR ABORT
;
: HI  . " Hello" CR ;
OB.ARRAY EVENTS
: INIT.EVENTS
   16 NEW: EVENTS ( allocate space for 16 CFAs )
   'C BAD.INDEX   FILL: EVENTS ( make safe )
   'C HI 3 TO: EVENTS
;
```

INCLUDE the file then test it by entering:

```
INIT.EVENTS
3 EXEC: EVENTS ( executes DOTHIS )
5 EXEC: EVENTS ( reports error )
FREE: EVENTS  ( frees memory now that we're done )
```

OB.ELMNTS

This class combines a two-dimensional array with the additional features of an ordered set of data. It has OB.ARRAY as a superclass. All of the methods that OB.ARRAY has, therefore, also apply to OB.ELMNTS. The rows of this array are called *elements.* The columns are called *dimensions.* This data structure can, therefore, be thought of as an ordered set of n-tuples. An example of this would be using OB.ELMNTS to represent X,Y,Z values. Each element would have 3 values, one for each physical dimension, X,Y and Z. Another example would be using an OB.ELMNTS to represent a melody. Then each point might have a Time and a Pitch value. The notes in this melody could be thought of as points in a 2 dimensional time/pitch space.

?NEW: (#elements #dimensions -- addr | 0)

Allocates memory for the data. The default width of each value is 4 bytes. If memory cannot be allocated a zero is returned.

ADD: (V1 V2 V3 … VN -- , adds an element to the end)

This adds one row, or element, to the end. The first ADD: goes into element number 0. The next ADD: goes into element number 1 and so on. See the tutorial for an example. It is *veryimportant* to have the right number of values on the stack when using ADD: . If you do not have the same number of values as the object is dimensioned, then you will have stuff left over on the stack or get stack underflows.

BACKWARD: (--)

Advances the cursor used by NEXT: backward one position.

CHOP: (index count --)

Remove count elements starting at index.

CURRENT: (-- V1 V2 V3 … VN , element at current position)

DIMENSION: (-- #dimensions)

Return number of dimensions declared.

DO: (function_cfa -- , pass each element to the function)

This is useful when you want to do something to each of the elements. The function must "eat" as many values as there are dimensions. For example, let's calculate the sum of the products of a 2 dimensional elements array.

```
10 2 NEW: ELM1
2 3 ADD: ELM1
4 5 ADD: ELM1
VARIABLE SUM-PRODS
:  *+EACH   (  a b -- , add product of a and b to SUM-PRODS )
    *  ( multiply the two values )
   SUM-PRODS +!   ( add result to variable )
;
0 SUM-PRODS !
0 GET: ELM1 *+EACH   ( -- , test function outside DO: )
SUM-PRODS @ .  ( should be 6 )
0 SUM-PRODS !
'C *+EACH  DO: ELM1    ( pass each element to function )
SUM-PRODS @ .  ( should be 26 )
FREE: ELM1
```

DUMP.SOURCE: (--)

Prints ODE source code that could be used to regenerate the state of this object. Combining this with LOGTO allows you to save the contents of an OB.ELMNTS array as source code. This code

can be reloaded using INCLUDE. Alternatively, you could use FWRITE and DATA.ADDR: to write a binary file containing the contents of an object. Be sure to put a header on the file that says how much to NEW: the object and how many data items and dimensions to use.

ED.AT: (element dimension -- value , return value at e,d)

Fetches value based on its row and column address.

ED.TO: (value element dimension -- , store value at e,d)

These two methods (ED.TO: and ED.AT:) are critical because all of the other access methods are written using them. If you need to define some special access methods for a new class based on OB.ELMNTS then you will probably need to use the methods ED.AT: and ED.TO: .

ED2I: (element dimension -- index , convert to linear index)

OB.ELMNTS are actually implemented using one dimensional arrays. This method returns the one dimensional array index for an element dimension pair.

EMPTY: (-- , set number of elements to zero)

This is faster then CLEAR: because CLEAR: actually sets the allocated memory to zero. This only sets the element counter to zero.

FILL.DIM: (value dim# -- , fill one dimension with a value)

FIRST: (-- V1 V2 V3 … VN , return first element, set pointer)

FOREWARD: (--)

Advances the cursor used by NEXT: foreward one position.

GET: (element -- V1 V2 V3 … VN , fetch a given element)

Using the example for the ADD: method:

```
1 GET: ELM1 .S ( will produce )
   777 888 999
```

GOTO: (element# --)

Sets the cursor used by NEXT: .

I2ED: (index -- element dimension , opposite of ED2I:)

I2ADDR: (index -- address , determine address of an item)

Calculates actual address of an item in the array. Use with ED2I: .

INSERT: (V1 V2 V3 … VN element -- , inserts new element before)

The pointers are adjusted as in REMOVE: to maintain current position.

LAST: (-- V1 V2 V3 … VN , return last element added)

MANY: (-- N , return number of elements added)

Sending a MANY: message to ELM1 (from the example for the ADD: method) returns a value of 2.

MANYLEFT: (-- N , number of elements left after current)

This is handy for checking whether an array has been exhausted using NEXT: .

MAX.ELEMENTS: (-- max , maximum number of elements allocated)

This will be the same as the first value passed to NEW: .

NEW: (#elements #dimensions --)

Calls ?NEW: and aborts if zero is returned. ?NEW: is preferred for use in applications.

NEXT: (-- V1 V2 V3 … VN , element at next position)

This also increments the internal pointer. This is useful for sequential processing. An error will occur if you attempt to go past the end of the array. Use MANYLEFT: to see how may are left.

NEXTWRAP: (-- V1 V2 V3 … VN , element at next position)

This also increments the internal pointer. This is useful for sequential processing. When the end of the array is reached, NEXTWRAP: will automatically wrap around back to the first element. Use WHERE: to find out where you are if need be.

PRINT: (-- , prints out elements in a table)

PRINT.DIM: (dim# -- , print the values of a single dimension)

PUT: (V1 V2 V3 … VN element -- , stores a complete element)

This stores a row without incrementing any internal pointers. It is usually used for editing.

REMOVE: (element -- , removes a given element)

When pointing to an element above the one that was removed, the internal pointers will be decremented so they still point to the same data. This moves all the higher elements down to fill in the gap.

RESET: (-- , Resets the current element pointer to 0)

This often precedes successive calls to NEXT: .

SET.MANY: (N --)

Set the number of elements currently in object. If you want to make this object act as if you have ADDed a bunch of values, you can set the number of elements it thinks it has. You can use SET.MANY: and then use GET: and PUT: to access any element within the range you set. For example, instead of adding elements:

```
OB.ELMNTS ELM-1
20 3 NEW: ELM-1
15 SET.MANY: ELM-1
PRINT: ELM-1    ( notice 15 elements )
FREE: ELM-1
```

Note that the value for SET.MANY: cannot be greater than that for which the object was NEWed.

SIZE: (-- N , return number of individual values added)

This is the product of MANY: and DIMENSION: .

SMEAR: (start count -- , smear element up)

Overwrites count elements with element at start.

SPLIT: (start count -- , move elements up and create a split)

This is used internally by INSERT: but can be used externally. Copies elements above and including start up by count. This is more difficult to explain than it is to see: try it.

STRETCH: (index count -- , dup element at index , count times)

This will "internally" duplicate a specific element in an object, count-number of times.

WHERE: (-- element#)

Returns the cursor used by NEXT: .

OB.LIST

This is a one dimensional version of OB.ELMNTS. It is handy for keeping a list of things. It has one new method and one altered method.

NEW: (#elements -- , allocate memory for one dimension)

The number of dimensions is automatically one.

DELETE: (value -- , removes value from list)

Looks for the value in the list and removes it if found. For example:

```
OB.LIST MY-LIST
10 NEW:  MY-LIST    ( make room )
1111 ADD: MY-LIST
234 ADD: MY-LIST
1988 ADD: MY-LIST
PRINT: MY-LIST   ( see all values )
234 DELETE: MY-LIST
PRINT: MY-LIST   ( 234 now gone )
FREE: MY-LIST
```

OB.OBJLIST

This class is used for storing the *addresses of other objects*. Its superclass is OB.LIST so all of those methods are inherited. OBJLIST's are extremely important in ODE, where you typically have lists of various objects that you want to be able to index into, rearrange, perform operations on, and so on.

?INSTANTIATE: (class_cfa #objects -- class_pfa | 0)

Dynamically instantiate the objects and store their addresses in the objlist. This will call ?NEW: first to make room for the object addresses. Returns zero if it could not allocate all of them. For more information on dynamic instantiation, see the advanced topic later in this chapter.

DEINSTANTIATE: (--)

Deinstantiates all objects created using ?INSTANTIATE. Then FREE: the objlist.

FREEALL: (--)

Send a FREE: message to all objects listed within. This can be very useful in FREE:ing all objects in a list, when you might not necessarily know exactly what those objects are! Note that FREEALL: does not actually FREE: the object list itself (just use FREE: for that).

Dynamic Instantiation using OB.OBJLIST

Here is an example that uses OB.OBJLIST to dynamically instantiate 25 arrays from memory. Put this code in a file so you can experiment with it.

```
OB.OBJLIST MUCHO-ARRAYS \ declare array to hold arrays!

:  MAKE.MUCHO-ARRAYS  ( -- error? )
\ make room for 25 arrays in the object list
   'C OB.ARRAY 25 ?INSTANTIATE: MUCHO-ARRAYS
   IF  \ yes we made them
      25 0
      DO
         I GET: MUCHO-ARRAYS  \ get Nth array
         10 OVER  ( -- array 10 array )
         NEW: []
      ( -- array , make room for 10 in each array )
         I SWAP   ( -- I array )
         FILL: [] ( -- , fill dynamic array with I )
      LOOP
      FALSE \ no error
   ELSE
      TRUE \ couldn't instantiate!!
   THEN
;

: CLEANUP.MUCHO-ARRAYS ( -- )
\ free memory in all arrays in object list
   FREEALL:  MUCHO-ARRAYS
   DEINSTANTIATE: MUCHO-ARRAYS
;
\ Automatically cleanup if file is forgotten.
IF.FORGOTTEN CLEANUP.MUCHO-ARRAYS
```

Now include the file, and enter:

```
MAKE.MUCHO-ARRAYS .
```

You will now have an object list, 25 long, which contains the addresses of the twenty-five arrays. If the instantiation failed, MAKE.MUCHO-ARRAYS would return a TRUE as an error flag.

Since these arrays have no names, only addresses in MUCHO-ARRAYS, you can only reference them by indexing into the list. This will generally involve the technique of late binding, since you will have to pass the address of the object to the method on the stack.

For example, if you wanted to PRINT: one of the dynamically instantiated arrays, here's how you would do it:

```
5 AT: MUCHO-ARRAYS  ( -- address-of-array#5 )
PRINT: []
```

When you are done using these arrays, you must free them and then deinstantiate them. Enter:

```
CLEANUP.MUCHO-ARRAYS
```

Defining New Classes and Methods

New classes can be defined if the existing ones do not provide the functionality that you need. You should first choose the closest available class to use as the *superclass*. Then decide what internal *instance variables* each object will need. Any methods that have not already been *declared* will need to be so that they will have a method index. Do this simple by saying:

```
METHOD new_method:
```

before your class definition (where new_method: is the name of your new method). If you are just giving new functionality to an old method you will not need to declare a new name. For example, if you modify the PRINT: method, you won't have to *declare* PRINT: as a method since it already been declared for other classes.

It is generally good programming practice to use the same method name for functions that are more or less the same. For example PRINT: for OB.INT does different things than PRINT: for OB.ARRAY, but the user only has to remember a single method name.

If you *instantiate an object inside a class definition*, then it will become an *instance object* that will exist inside every instance of the new class. See the later example of Instance Objects.

Class Definition Glossary

The following words can be used in the definition of new classes. It is easiest to learn them from the examples.

:CLASS (<word> -- , declare a brand new class)

:M (<word> -- , start the definition of a method, like :)

;CLASS (-- , terminates a class' definition)

;M (-- , terminates a methods definition)

<SUPER (<word> -- , declare the superclass of a class)

The new class will *inherit* all of the *instance variables* and *methods* of the superclass. This word is required for every class definition. If there isn't any special existing class that you would like to inherit from then just use OBJECT .

IV.LONG (<name> --)

Creates a named 32 bit instance variable.

IV.SHORT (<name> --)

Creates a named 16 bit instance variable.

IV.BYTE (<name> --)

Creates a named 8 bit instance variable.

IV.BYTES (count <name> --)

Make room for "count" bytes in object. The other words, (IV.LONG, IV.SHORT, IV.BYTE, etc.) were created using BYTES. All instance variables must be declared *before* any method definitions.

METHOD (<word> -- , declare word to be a new method)

It is recommended that the method declared end in a colon to be consistent with the syntax of other object oriented languages. Methods only need to be declared once and can then be redefined for several different classes.

It is necessary to declare a method before the class is defined because the number of methods declared determines how large to make the classes method table. Each class has a table of CFAs, one for each declared method. Each method is assigned an index in that table. This technique allows for extremely fast binding between messages and method code at the expense of some memory. ODE requires more memory for class definitions than other similar systems, but executes late binding much faster. This is a big advantage in real time applications like games or music. Plus memory keeps getting cheaper so why not use it.

Instance Variables

The following words are used inside method definitions to access the instance variables of an object. When these words are executed they calculate the address of the instance variable data, then fetch that data. The fetch will automatically be of the proper width, for example, an IV.SHORT will use W@. In this way, different objects can have their own private copies of the data. To store data into these instance variables you must use one of the prefix operators described below. [Technical note: The address of the current object is kept on a special stack called the *object stack*. When a method starts to execute, it can be assumed that that object's address is at the top of the object stack. The instance variables are defined in terms of their offsets from the address of that object].

IV=> (value <instance_var> --, Store value in ivar)

This word will look up the width of the instance variable and use the appropriate store function, C!, W! or ! . To fetch the value of an instance variable just give its name. As an example, if you want to set an instance named IV-DEPTH to 200:

```
200  IV=>  IV-DEPTH  ( only valid inside a method )
." Default depth = " IV-DEPTH  .  CR
```

IV+> (value <instance_var> -- , Add value to ivar)

Note: This only works on long variables, ie. those declared with IV.LONG.

IV&> (<instance_var> -- addr , Address of ivar)

This is useful if you need an indexed instance variable or if you need to pass an instance variable's address. Generally, if you are doing a lot of indexing of instance variables, it is probably better to use *instance objects*.

Using SELF in Method Definitions

Sometimes, to define a new method, you will need to use other methods already defined for that class. This creates a problem: what object do you pass the message to inside the method definition? The usual technique of METHOD: OBJECT can't work because the objects themselves haven't been defined yet, so there is no address of any object to use for the method. Smalltalk and ODE solve this problem with two special words, SELF and SUPER . SELF allows you to refer to whatever object is currently being defined inside a method definition for that object. For example, when you see, CLEAR: SELF inside of a method definition you know that the object currently being called will be cleared. SELF and SUPER can only be used inside a method definition.

Warning: methods referenced using method_name: self must already be defined. Otherwise any previous definition of that method for that class will be used. This is consistent with the Forth convention of not allowing forward referencing.

For example, let's define a method called DIM.SUM: for a class which is a subclass of OB.ELMNTS. DIM.SUM: will sum the values for a given dimension. (Note that we use local variables in the method. The '{' denotes local variables instead of a normal stack diagram. See the JForth or HForth manual for more information on local variables.) Enter this in a file using a text editor.

```
\ Declare method to be defined.
METHOD DIM.SUM:

\ Declare new class as a subclass of OB.ELMNTS
:CLASS  OB.NEW.ELMNTS  <SUPER OB.ELMNTS

:M DIM.SUM: {  dim | cursum -- sum-of-all-values }
   0 -> CURSUM
   MANY: SELF 0
   DO
      I DIM ED.AT: SELF
      +-> CURSUM
   LOOP
   CURSUM
;M
;CLASS
\ Now let's instantiate one and test it.
OB.NEW.ELMNTS  NELM-1
: TEST.NELM
   10 2 NEW: NELM-1
   5 20 ADD: NELM-1
   5 7 ADD: NELM-1
   ." Sum of dimension 1 = " 1 DIM.SUM: NELM-1 . CR
   FREE: NELM-1
;
```

To test it, INCLUDE the file and enter:

```
TEST.NELM
```

Using SUPER and SUPER-DOOPER in Method Definitions

SELF is used more often than SUPER. SUPER is only needed when the new class redefines a method but you still want to access the old one. For example, if you want a method that prints an object as previously defined but also prints out some dashes, and an item-count:

```
:M PRINT: ( -- , extended print )
   PRINT: SUPER
   CR ." --------------" CR
   MANY: SELF . ." items" CR
;M
```

In this example, using PRINT: SELF would have resulted in a fatal recursion — you can't use a method that is currently in the midst of being defined!.

Occasionally you won't like the way your superclass performs a method, but you like the way that superclass's superclass does. In this situation you can use SUPER-DOOPER instead of SUPER.

```
:CLASS OB.STRANGE <SUPER OB.ELMNTS
:M PRINT:   ( -- )
   ." Double Print!!!" CR
   PRINT: SUPER-DOOPER ( print like an ob.array )
   ." ----------------------" CR
   PRINT: SUPER ( print like an ob.elmnts )
;M
;CLASS
```

SUPER-DOOPER is not really standard object oriented programming, it's something we added to ODE when we needed it in the implementation of HMSL, and it has proved useful ever since.

Special Methods: INIT:

The method INIT: is automatically executed when an object is *instantiated*. If you want to set *default values for the instance variables*, or perform any *special initialization*, just define an INIT: method. You will probably want to include an INIT: SUPER to invoke any initialization that the superclass was doing. Generally it's best to avoid doing things inside INIT: that affect the state of the operating system. This includes allocating memory (NEW:), opening files, etc. Otherwise you will not be able to save the INITed object in a precompiled form, for its pointers to memory and files will be invalid the next time it is run.

Example Class Definition

To clarify this, let's define a class of object that is a city. We want to keep track of population and area. Let's use the OB.INT class as the superclass since it already does some of what we want the new class to do. We can use the existing instance variable in OB.INT for the population. We will need to add a new instance variable for the area.

```
METHOD PUT.AREA: ( declare new methods )
METHOD GET.AREA:
METHOD CALC.DENSITY:

:CLASS OB.CITY <SUPER OB.INT  ( inherit properties of OB.INT )
   IV.SHORT IV-AREA ( declare 2 bytes for the area )
```

```
:M INIT: ( -- , Set defaults )
   INIT: SUPER ( perform initialization of superclass, important )
   1 IV=> IV-AREA ( avoid 0 divide in CALC.DENSITY: )
;M

:M PUT.AREA: ( area -- , set area of city )
   IV=> IV-AREA ( store using prefix operator )
;M

:M GET.AREA: ( -- area , return city's area )
   IV-AREA ( leaves value automatically )
;M

:M CALC.DENSITY: ( -- density, return people per acre )
   GET: SELF GET.AREA: SELF /
;M

:M PRINT: ( -- , redefine print to perform new function )
   ." The city of " NAME: SELF CR
   ." has a population of " GET: SELF . CR
   ." and an area of " GET.AREA: SELF . CR
   ." The density is " CALC.DENSITY: SELF . CR
;M
;CLASS ( finish class definition )

OB.CITY ARMPIT ( declare two instances )
OB.CITY BIGHOLE

10034  PUT: ARMPIT ( set populations )
795  PUT: BIGHOLE

45  PUT.AREA: ARMPIT ( set areas )
29  PUT.AREA: BIGHOLE

PRINT: ARMPIT ( print city reports as defined )
PRINT: BIGHOLE
```

Example of Creating a Class with Instance Objects

Lets create a subclass of OB.ELMNTS that keeps information about each of it's dimensions. It will have an internal array with one item per dimension. It will be sent a NEW: and FREE: message as part of the class' NEW: and FREE: method.

```
( declare methods for accessing Instance Object)
METHOD PUT.DIM.DATA:
METHOD GET.DIM.DATA:
:CLASS FOO  <SUPER OB.ELMNTS
```

```
    IV.LONG IV-FOO-FLAVOR   ( normal instance variable )
    OB.ARRAY IV-FOO-ARRAY  ( Declare Instance Object )

:M NEW: ( #elmnts #dimensions -- )
    TUCK NEW: SUPER    ( calls FREE: )
    NEW: IV-FOO-ARRAY  ( so NEW: this after SUPER )
( If you reverse the order above, IV-FOO-ARRAY will get FREE:d )
( after NEW: SUPER because NEW: SUPER calls  SELF FREE: [] )
;M

:M FREE: ( -- , called by NEW: )
    FREE: SUPER
    FREE: IV-FOO-ARRAY
;M

:M PUT.DIM.DATA: ( data dim -- )
   \ set associated data for dimension
    TO: IV-FOO-ARRAY
;M
:M GET.DIM.DATA: ( dim -- data , get associated dim data)
    AT: IV-FOO-ARRAY
;M

:M PRINT:  ( -- print values and dim data )
   PRINT: SUPER
   ." Dimension values ---" CR
   PRINT: IV-FOO-ARRAY
;M
;CLASS

\ Instantiate and use a FOO
FOO MY-FOO
20 3 NEW: MY-FOO
11 22 33 ADD: MY-FOO
234 345 456 ADD: MY-FOO
77 1 PUT.DIM.DATA: MY-FOO    ( set dimension value )
PRINT: MY-FOO
FREE: MY-FOO
```

Remember, instance objects, like instance variables, can *only* be referenced from inside a method or a Forth word called from a method for that object!

Advanced Topics

ODE Functions

CURRENT.OBJECT (-- object)

Object currently being processed. This word is handy if a method calls a function and the function needs to know which object called it The function can then send late bound messages to the object. A lot of HMSL objects, like jobs and interpreters automatically pass their own addresses to many of their methods, so CURRENT.OBJECT is only necessary in unusual cases.

RUN.FASTER (--)

Turn off some error checking for speed. Turns off late bound class checking which ensures that you are sending messages to a real object and not some random piece of memory. Also turns off range checking for subsequently compiled array based classes. Only call this when you have finished debugging your program. Recompile your aplication after calling this.

RUN.SAFER (--)

Turns on error checking turned off by RUN.FASTER Default mode for ODE.

Getting Information About Classes

There are several words that you can use to find out information about a particular class. These are useful if you want to get the most use from a given class.

METHODS.OF (<class> --)

Show the methods supported by a class. For example, to see what methods are supported by the OB.ELMNTS class, enter:

```
METHODS.OF OB.ELMNTS
```

INHERITANCE.OF (<class> --)

Show the superclasses of a class To see what classes OB.ELMNTS inherited its methods and instance variables from, enter:

```
INHERITANCE.OF OB.ELMNTS
```

ALL.METHODS (--)

Show all methods declared in dictionary.

Dynamically Allocated Objects

Sometimes you may want to create, or *instantiate,* an object while a program is executing. Otherwise all objects would have to be defined at compile time. There are two words used in dynamic instantiation:

?INSTANTIATE (<class> -- object-address | 0)

This creates an object of the requested class in free memory. The object is added to a list of dynamic objects where it can be found using 'O . It will be given the name DYNn, where nnn goes from 000 to however many you have. You can't reference these by name since there will be no NFA (name

field address) to use for the name. You can change the name using PUT.NAME: but this is only used for printing purposes, and cannot be used for messaging. All messages to this object must be sent using late binding, and for this you would typically retrieve the address of the object from some user created object list or array (see the example with OB.OBJLIST).

DEINSTANIATE (object-address --)

Deallocate dynamically instantiated object.

Here is an example of a dynamically allocated array object. It uses a local variable to store the object address. Binding to a local variable is always late binding. Notice that in this example we check to make that INSTANTIATE and ?NEW: returned address indicating success before proceding.

```
: DYNARRAY { newobj -- , Demo dynamic array. }
   ?INSTANTIATE OB.ARRAY \ Create dynamic object
   DUP -> NEWOBJ  \ save in local
   IF
\ Allocate data space, note use of late-binding
      12 ?NEW: NEWOBJ
      IF
         789 FILL: NEWOBJ
         PRINT: NEWOBJ
         FREE: NEWOBJ  \ Free allocated memory
      THEN
      NEWOBJ DEINSTANTIATE \ Deallocate object.
   THEN
;
Now test this by entering:
DYNARRAY
```

'O (<dynamic-object> -- object-address)

Search the list of dynamically instantiated objects for the object with the matching name.

If you need to use an array inside of another object, you could instantiate one and store its address in an instance variable. However, you can also declare instance objects (described previously in this chapter).

Examining Instance Variables

When debugging object-oriented code, it is useful to be able to examine instance variables even if there is no method for accessing them. To do this you can use the DUMP: method. ODE supplies an alternative that is “technically” illegal in an object-oriented system and allows the user to reference an instance variable using techniques normally used only for *C* structures. In the previous example, if you did not have a method for GET.AREA: , you could enter:

```
ARMPIT REL->USE ..@ IV-AREA . ( Get the area of Armpit )
```

Don’t try this at home, kids.

Error Reporting

There is a special facility for reporting errors inside objects. It dumps the object stack which gives a traceback of which objects were calling which.

OB.REPORT.ERROR ($method-name $message severity --)

The severity is either ER_WARNING , ER_RETURN , or ER_FATAL . For fatal errors, the object stack is printed, cleared, and execution aborted. Here is an example of detecting an out of range error in a method called DOIT:

```
" DOIT:"  " Input out of range"
ER_FATAL  OB.REPORT.ERROR
```

Inheritance

A class definition contains several things used when one class inherits instance variables and methods from another class. The number of bytes of instance variable space is kept. When the subclass has instance variables defined, their offset begins after the superclass's area.

A table of CFAs for the methods of a given class is kept at the end of a class definition. Each declared method has a *method index* that is the same for all classes. When a new class is defined, a table large enough to hold all of the declared methods is alloted in the dictionary. The word <SUPER copies the CFAs from the superclass into the CFA table of the subclass. A message sent to an object will use these CFAs unless a new method has been defined that overwrites that entry in the table. When SUPER is used, the CFA from the superclass's table is compiled, regardless of the value in the current class CFA table.

Memory Placement for Amiga

If you create an array object that will be used by the Amiga coprocessors, then the data must be allocated in chip memory. This can be controlled by setting a variable called MM-TYPE to MEMF_CHIP before calling NEW: .

```
OB.BARRAY  IMAGE-DATA
MM-TYPE @ ( Save old value )
MEMF_CHIP MM-TYPE !
32 NEW: IMAGE-DATA ( Allocate space in CHIP RAM )
MM-TYPE !
```

This is done automatically for HMSL classes that need it, i.e. OB.WAVEFORM, OB.SAMPLE and OB.ENVELOPE.

Cloning ODE Programs using JForth

If you are going to Clone a JForth program that uses ODE, you MUST do the following:

1) Compile REDEFs that are needed by Clone. These are loaded by default if you use the file LOAD_ODE. If not you MUST:

```
INCLUDE JO:CLONE_SUPPORT
```

2) Do all memory allocation at Run Time. This means you **cannot** call NEW: at compile time. You must call it from a word in your program if you need it. If you can save a program using SAVE-FORTH then you are OK as far as this requirement is concerned.

3) Initialize the Object Stack. The pointer uses absolute addresses for speed and must be converted before running ODE. At the beginning of your program, therefore, you must call **OS.SP!** or you will definitely crash.

4) We recommend that you compile Clone before compiling ODE but it is not required.

5) Since name fields are not in a Cloned image, if you are going to use NAME: or PRINT: then you must give the object a name explicitly using PUT.NAME:.

Here is an example of an ODE program that will work with Clone.

```
INCLUDE? TASK-CLONE_SUPPORT  JO:CLONE_SUPPORT
OB.ARRAY  MY-AR1
: GOOD.ODE  ( -- simple clonable program )
    OS.SP!   ( REQUIRED!!!!! )
    10 NEW: MY-AR1   ( only call NEW: at run time )
    " TestArray" PUT.NAME: MY-AR1    ( since NFAs will be gone )
    761 FILL: MY-AR1
    PRINT: MY-AR1
    FREE: MY-AR1
;
( now clone it )
CLONE GOOD.ODE
SAVE-IMAGE GOOD.ODE  RAM:GOOD.ODE
( now in the CLI, enter )
RAM:GOOD.ODE
```

Explanation of ODE Structures Diagram

This is a *very* technical discussion of how ODE is implemented and is probably more detailed than most people need to know.

A class structure contains information about how to create an object and how to implement its methods.. Please refer to the accompanying diagram when reading this section. The first field in the class structure is the Size of an Object. When an object is instantiated, this much room is alloted or allocated. The second field is the Number of Methods that this class supports. This determines how large the jump table containing method addresses is. The next field, the Validation Code, is used to distinguish valid classes from random memory and is used for error checking when binding. The next field is the Superclass pointer.

When a new class is defined the following things occur. The new class first inherits the superclass' initial object size. As new instance variables are declared, this size is increased. A jump table is alloted for the class that has enough entries for all of the methods declared using METHOD . The superclass' method pointers are copied from the superclass jump table so that the new class can inherit those methods. When new methods are defined, the entry in the jump table is overwritten. Each declared method has a specific index which determines its offset in any class' jump table.

When an object is instantiated, space is alloted based on the Size of Object field in the class. The first cell of the object is set with a pointer to the method jump table for the class. Then the first method in that jump table, INIT: , is called which initializes the object.

When a message is sent to an object, the object's address is first pushed onto the "current object stack". Then the appropriate method is looked up in the object's classes jump table and executed. When finished, the object is popped from the object stack. This allows nesting of object method calls.

When a method is declared using METHOD a structure is created that has an index equal to the current value of MI-NEXT. MI-NEXT is then incremented. This is the index for a specific method in each

class' jump table. The methods are linked using the Previous field so that the METHODS.OF word can scan a method list for a class.

Chapter 11
Miscellaneous Forth Tools

Memory Allocation

History

Historically, all memory allocation in traditional Forth systems has been from the available dictionary space, the same area that new definitions are added to. While this is sometimes a preferred technique, it is often lacking as the sole source of pooled memory.

With the advent of sophisticated operating systems such as that available on the Amiga, Forth may inherit the ability to 'request' memory from a free memory pool maintained at no burden to the application programs.

JForth provides a rich set of operators that fully interface with the Amiga memory manager, and additionally provide capabilities specifically suited to the Forth environment.

Allocate and Free Memory.

JForth provides two basic primitives for handling memory requests...

ALLOCBLOCK (type size -- memblk | FALSE)

Allocate from other than the available dictionary, a block of memory of 'size' number of bytes. The type may be appropriate combinations of: MEMF_PUBLIC, MEMF_CHIP, MEMF_FAST, MEMF_CLEAR and MEMF_LARGEST. Returns false if AllocMem() failed. Refer to the AmigaDOS ROM Kernel Manual for details on the Amiga memory map and use of the 'MEMF_' constants. NOTE: a type=0 is default PUBLIC, FAST if available.

FREEBLOCK (memblk --)

Frees the memory, making it available again. If you pass the address of memory that has already been freed then the Amiga will probably crash.

FREEVAR (var-addr --)

If the contents of the variable is non-zero, then the contents is passed to FREEBLOCK then the variable is set to zero. This is a handy word when you store the address of allocated memory in a variable.

```
VARIABLE MY-VAR
MEMF_CLEAR  1000 ALLOCBLOCK   \ allocate memory
DUP .  \ is it zero?
MY-VAR !  \ save address in variable
MY-VAR FREEVAR  \ now free it
```

The address returned from ALLOCBLOCK is a JForth relative address, and as such, it is legal to use the Forth memory access words (@, !, etc) on them.

The programmer may allocate memory with ALLOCBLOCK, and use it as desired. When he no longer needs the area, he should call FREEBLOCK to return it for other other tasks to use.

Note that allocated blocks will not be preserved across the SAVE-FORTH command or when CLONING. Your program should allocate blocks and free them at run-time. You should avoid calling ALLOCBLOCK while compiling. For large areas that are too difficult to build at run-time (such as SINE or TRIG tables), we recommend storing them in a file, and loading them at run-time. See the chapter on the JForth File Manager.

Use Memory like a Stack.

Certain words provided by JForth are available to perform automatic storage into and out of these areas. These words maintain a 32-bit counter that is allocated along with each memory area; this counter is used to reflect the current amount stored in the memory block (when these words are used).

FREEBYTE (memblk -- next-free-byte-pos)

Returns the contents of the counter. This counter is set to zero when the memory block is allocated.

FREEBYTEA (memblk -- address-of-freebyte-counter)

This function returns the address of the 'freebyte' counter. The programmer may make this counter available for local storage simply by not using the words that update it. This way, he may write to his area as desired, storing into the FREEBYTE counter to save, for example, how much data is there.

SIZEMEM (memblk -- allocated-size)

Return the originally allocated size of this block of memory.

The words that automatically update the FREEBYTE counter are concerned with utilizing the memory-block as a STACK. These first two implement it as a simple last-in/first-out buffer:

PUSH (n1 memblk --)

Push 'n1' on the stack at 'memblk'.

POP (memblk -- n1)

Retrieve the last item 'pushed' on this stack.

Note that, for efficiency reasons, error-checking has not been implemented in PUSH and POP..if there is a chance your stack could become full or empty, you should check the FREEBYTE contents and compare it with SIZEMEM before each PUSH or that it is not zero before a POP.

The next two words also use the area as a stack, but operate somewhat differently. Firstly, they do not expect the address of the area but a location CONTAINING the address.

+STACK (n1 var-addr --)

Push 'n1' on the stack whose address is contained in the var-addr. Note that if the variable holds zero, the memory-block will be allocated, and the address placed in the variable.

-STACK (n1 var-addr --)

Search for an occurrence of 'n1' on the stack whose address in contained in the var-addr. If found, remove the occurrence and return. If the number of elements on the stack becomes zero during this call, the memory block is de-allocated via FREEBLOCK, and the Var-addr will be cleared.

Note that error-checking has been implemented in +STACK and -STACK. +STACK will automatically expand the stack memory area if necessary (and change the contents of VAR-ADDR in the process). -STACK will deallocate the stack memory and clear VAR-ADDR if it removed the last stack item.

Example.

Allocate 1K of CHIP memory, accessible by anyone...

```
: GET1K  ( -- memblk )
    MEMF_PUBLIC  MEMF_CHIP  OR     1024
    ( -- type size ) ALLOCBLOCK ?DUP 0=
    IF   ." Memory allocation failed!"  QUIT
    THEN
;
VARIABLE  MY-MEM
GET1K MY-MEM !  ( allocate and save pointer )
```

We can now use this memory for any purpose.

```
2345 MY-MEM @ !  ( set first cell in block )
77 MY-MEM @ 100 + ! ( set cell at offset 100 )
```

Alternatively, we can this area as a stack. Let's push 11, 22 and 33 to 'MY-MEM'...

```
11 MY-MEM @ PUSH
22 MY-MEM @ PUSH
33 MY-MEM @ PUSH
765 MY-MEM @ PUSH
MY-MEM @ POP . ( print 765 )
```

Now selectively remove the '22' ...

```
22  MY-MEM -STACK
```

The MY-MEM stack now has (-- 11 33) We're all done, so return what we've borrowed...

```
MY-MEM @  FREEBLOCK
```

Deferred Words

The DEFER facility allows you to call a word before it is actually defined. You can then define and redefine that word without recompiling the code that calls it. This is very useful if you are testing a word that is compiled at the beginning of a large program. It is also useful for redefining system words that you cannot recompile, like KEY , EMIT , and FIND .

Using DEFER to "vector" code.

Let's look at how we can use DEFER to experiment with a word. Enter the following at the keyboard:

```
DEFER DOIT    ( create a deferred word )
DOIT   ( executes QUIT )
: USEIT  ( -- , use DOIT in a loop )
    8 0 DO  DOIT  LOOP
;
USEIT
```

You will notice that nothing appears to happen when you enter DOIT or USEIT. This is because the default action for a deferred word is QUIT. Now let's try changing what DOIT does without recompiling USEIT . Enter:

```
: HI ." Hello" CR ;
HI
' HI IS DOIT
DOIT
```

Notice that DOIT is now the same as HI. The Forth word IS takes a CFA from the stack and places it inside the deferred word for use when it is called. You can now enter USEIT and see that its execution has been changed without recompiling. This process of using a word to point to another word is called "vectored execution".

You can use WHAT'S to find out what a deferred word is defined as. Enter:

```
WHAT'S DOIT >NAME ID.
```

Deferred System Words

Many JForth system words are deferred (vectored) so that you can change how they work. This allows you to do things like redirect I/O and extend the compiler. These words are maintained by the JForth system. (See FREEZE).

Here is a partial list of JForth words which are deferred:

```
EMIT        KEY         ?TERMINAL   CR          QUIT
INTERPRET   FIND        :CREATE     NUMBER      ID.
SOURCE      BLOCK       'WORD       LONGCFA,    CFA,
FLUSHEMIT   COLDEXEC    ABORT       EXPECT      TYPE
```

To create a stuttering EMIT (2 emits/char) , we could try:

```
DEFER OLD-EMIT
```

DEFER creates a new deferred word called OLD-EMIT .

```
: EEMMIITT  ( char --- )
    DUP OLD-EMIT OLD-EMIT
;
: STUTTER   ( --- )
    WHAT'S EMIT  ( get the current value of EMIT )
    IS OLD-EMIT  ( save this value in OLD-EMIT )
    ' EEMMIITT IS EMIT
;
: STOP-IT!  ( --- )  WHAT'S OLD-EMIT IS EMIT  ;
STUTTER  .S
```

Things are going to be hard to read nnooww. After a while you'll get tired of tthhiiss ,, so enter:

```
STOP-IT!   ( this will appear as SSTTOOPP--IITT!! )
```

DEFER and GLOBAL-DEFER are functionally identical in JForth.

We recommend that you use a consistent naming convention for the words that you set the deferred words to. Brackets and parentheses are common, for example:

```
<ROBOT.OUTPUT>   or  (FILE.OUTPUT)
```

If you have a complicated program with a lot of deferred words, you might put in a word to display the state of all your important deferred words:

```
: SHOW-ME
    WHAT'S OUTPUT.FUNCTION >NAME ID.  CR
    WHAT'S INPUT.FUNCTION  >NAME ID.  CR
;
```

Potential Problems with Defer

Deferred words are very handy to use, however, you must be careful with them. Suppose you change EMIT so that it calls your word. Then suppose you forget your word that EMIT now calls. As you compile new code you will overwrite the code that EMIT calls and it will crash miserably. You MUST reset any deferred words that call your code before you FORGET your code. The easiest way to do this is to use the word IF.FORGOTTEN to specify a cleanup word to be called if you ever FORGET the code in question. In the above example using EMIT , we could have said:

```
IF.FORGOTTEN  STOP-IT!
```

Another problem that can occur is if you initialize a deferred system more than once. In the above example, suppose we called STUTTER twice. The first time we would save the original EMIT vector in OLD-EMIT and put in a new one. The second time we called it we would take our new function from EMIT and save it in OLD-EMIT overwriting what we had saved previously. Thus we would lose the original vector for EMIT . You can avoid this if you check to see whether you have already done the defer. Here is a rewritten form of the above example that does this check.

```
DEFER OLD-EMIT
' QUIT  IS OLD-EMIT  ( set to known value )
: EEMMIITT  ( char --- , our fun EMIT )
    DUP OLD-EMIT OLD-EMIT
;
: STUTTER   ( --- )
    WHAT'S OLD-EMIT  ' QUIT =  ( still the same? )
    IF  ( this must be the first time )
        WHAT'S EMIT  ( get the current value of EMIT )
        IS OLD-EMIT  ( save this value in OLD-EMIT )
        ' EEMMIITT IS EMIT
    ELSE ."  Attempt to STUTTER twice!" CR
    THEN
;
: STOP-IT!  ( --- )
    WHAT'S OLD-EMIT ' QUIT =
    IF  ." STUTTER not installed!" CR
    ELSE  WHAT'S OLD-EMIT IS EMIT
        ' QUIT IS OLD-EMIT  ( reset to show termination )
    THEN
;
IF.FORGOTTEN  STOP-IT!   ( make sure we clean up )
```

In the above example, we could call STUTTER or STOP-IT! as many times as we want and still be safe. Look in the files JU:LOGTO, JU:DEBUGGER, and JU:HISTORY for examples of code that sets deferred system words in a safe manner.

Finally, it is important to insure that, whenever you install a word into a deferred function, the word being installed does not directly call that deferred function. Doing so will create infinite recursion (actually, it will only recurse until it crashes). For example, since the word . (dot) normally calls EMIT, you cannot:

```
' . IS EMIT   ( never do this )
```

That would cause EMIT to call . which will call EMIT which will call . which will...get the idea?

Tools for FORGET

When you are testing a file full of code, you will probably recompile many times. You will probably want to FORGET the old code before loading the new code. You could put a line at the beginning of your file like this:

```
FORGET XXXX-MINE     : XXXX-MINE ;
```

This would automatically FORGET for you every time you load. Unfortunately, you must define XXXX-MINE before you can ever load this file. We have a word that will automatically define a word for you the first time, then FORGET and redefine it each time after that. It is called ANEW and can be found at the beginning of most JForth files. We use a prefix of "TASK-" followed by the filename just to be consistent. This TASK-name word is handy when working with INCLUDE? as well. Here is an example:

```
\ Start of file
INCLUDE? TASK-LOGTO JU:LOGTO
ANEW TASK-THIS-FILE
\ the rest of the file follows...
```

Notice that the INCLUDE? comes before the call to ANEW so that we don't FORGET LOGTO every time we recompile.

FORGET allows you to get rid of code that you have already compiled. This is an unusual feature in a programming language. It is very convenient in Forth but requires care. Most problems with FORGET involve leaving addresses that point to the forgotten code that are not themselves forgotten. This can occur if you set a deferred system word to your word then FORGET your word. (See the section above on DEFERred words)

Another problem is if your code allocates memory, opens files, or opens windows. If your code is forgotten you may have no way to free or close these things. You could also have a problem if you add addresses from your code to a table that is below your code. This might be a jump table or data table.

Since this is a common problem we have provided a tool for handling it. If you have some code that you know could potentially cause a problem if forgotten, then write a cleanup word that will eliminate the problem. This word could UNdefer words, free memory, etc. Then tell the system to call this word if the code is forgotten. Here is how:

```
: MY.CLEANUP  ( -- , do whatever )
    MY-MEM @ ?DUP
    IF  FREEBLOCK   0 MY-MEM !
    THEN
```

```
;
IF.FORGOTTEN  MY.CLEANUP
```

Notice that the cleanup word checks before doing FREEBLOCK. This word could be called any number of times and only the first time will have an action. (Otherwise you would crash the second time.)

IF.FORGOTTEN creates a linked list node containing your CFA that is checked by FORGET. Any nodes that end up above HERE after FORGET is done are executed.

Sometimes, you may need to extend the way that FORGET works. FORGET is not deferred, however, because that could cause some real problems. Instead, you can define a new version of [FORGET] which is searched for and executed by FORGET. You MUST call [FORGET] from your program or FORGET will not actually FORGET. Here is an example.

```
: [FORGET]  ( -- , my version )
    ." Change things around!" CR
    [FORGET]  ( must be called )
    ." Now put them back!" CR
;
: FOO ." Hello!" ;
FORGET FOO
```

This is recommended over redefining FORGET because words like ANEW that call FORGET will now pick up your changes.

Local Variables

The code for Local Variable support is in the file JU:LOCALS . If you want to use locals in your program, place this line at the beginning of your file:

```
INCLUDE?  {  JU:LOCALS
```

In a complicated Forth word it is sometimes hard to keep track of where things are on the stack. If you find you are doing a lot of stack operations like **DUP SWAP ROT PICK** etc. then you may want to use *local variables*. They can greatly simplify your code.

You can declare local variables for a word using a syntax similar to the stack diagram. These variables will only be accessible within that word. Thus they are "local" as opposed to "global" like regular variables. Local variables are *self-fetching*. They automatically put their values on the stack when you give their name. You don't need to @ the contents. Local variables do not take up space in the dictionary. They reside on the return stack where space is made for them as needed. Words written with them can be reentrant and recursive. You can declare how large local variables are to allow double precision, or larger.

Consider a word that calculates the difference of two squares, Here are two ways of writing the same word.

```
: DIFF.SQUARES ( A B -- A*A-B*B )
    DUP *
    SWAP DUP *
    SWAP -
;
  ( or )
```

```
: DIFF.SQUARES { A B -- A*A-B*B }
    A A *
    B B * -
;
3 2 DIFF.SQUARES  ( would return 5 )
```

In the second definition of DIFF.SQUARES the curly bracket '**{**' told the compiler to start declaring local variables. Two locals were defined, A and B. The names could be as long as regular Forth words if desired. The "**--**" marked the end of the local variable list. When the word is executed, the values will automatically be pulled from the stack and placed in the local variables. When a local variable is executed it places its value on the stack instead of its address. This is called *self-fetching*. Since there is no address, you may wonder how you can store into a local variable. There is a special operator for local variables that does a store. It looks like **->** and is pronounced "to".

Local variables need not be passed on the stack. You can declare a local variable that is uninitialized by placing it after a "vertical bar" (**|**)character. Uninitialized are exactly that. They are not automatically set to zero when created and may have any possible value before being set. Here is a simple example that uses **->** and **|** in a word:

```
: SHOW2*
   { loc1 | unvar --  , 1 regular, 1 uninitialized }
   LOC1  2*  ->  UNVAR
      (set uninitialized local to 2*LOC1 )
   UNVAR   .   ( print UNVAR )
;
3 SHOW2*   ( pass only 1 parameter, prints 6 )
```

Since local variable often used as counters or accumulators, we have a special operator for adding to a local variable It is +-> which is pronounced "plus to". These next two lines are functionally equivalent but the second line is faster and smaller:

```
ACCUM   10 +   -> ACCUM
10 +-> ACCUM
```

If you don't specify the size of the local variables, they default to 1 cell. You can specify the size in cells by preceeding the variable name with a number.

```
: EXAMPLE { 2 a 2 b 4 c -- d1 }
    a b  D+
    c ( note this fetches 2 double numbers )
    D+ D+ ;
5. 4. 6. 9. example  ( will return 24. )
```

Local variables are normally self fetching, but that can be turned on and off using NO@ and YES@. When NO@ is specified, a reference to a local variable will result in its address being placed on the stack.

```
: EXAMPLE { a b c -- sum+1 }
    a b +   +-> c
    no@ ( turn self fetching off )
    1 c +!
    yes@ ( turn it back on )
    c  ( fetch C ) ;
```

```
1 2 3 example ( will return 7 )
```

By combining NO@ and multiple cell variables, you can allocate local arrays

```
: LOCARR { indx | 100 larr -- }
   NO@ LARR YES@ \ get address of base of array
   INDX CELLS \ calculate offset into array
   +  @  \ calculate address in array and fetch value there
;
```

You can also specify that some local variables will automatically be returned:

```
: EXAMPLE { a b c --> c }
    a b + c + -> c
;
1 2 3 example  ( will return 6 )
```

The --> means that what follows up to the ASCII } is a list of variables to return.

If you name a local variable the same as a Forth word in the dictionary, eg. INDEX or COUNT, you will be given a warning message. The local variable will still work, but the earlier defined word of the same name will not be accessible until the ; at the end of the definition is reached. Other errors that can occur include, missing a closing '}', missing '--', or having too many local variables.

Logging to Files or the Printer

The output of JForth can be made to echo to a file as well as the keyboard by changing the deferred word EMIT. This is useful for generating records of your work, dumping memory to a file for later analysis, etc. If you are trying to generate formatted data files, you can get the code to work properly on the screen first. Then just log your output to a file. By sending the output to the file "PRT:", you can get JForth to send it's output to the printer. The words needed to do this are defined in the utility file called "JU:LOGTO".

```
$LOGTO  ( $filename -- , Send copy of output to file.)

LOGTO  ( <filename> -- , Get filename and pass to $LOGTO)

LOGSTOP  ( -- , Temporarily stop echoing. )

LOGSTART  ( -- , Continue echoing, used after LOGSTOP )

LOGEND  ( -- , Stop echoing, close file opened by LOGTO )

PRINTER.OFF  ( -- , turn off printer echo )

PRINTER.ON  ( -- , echo to printer as file PRT: )
```

As an example, if you want to dump some memory and have it printed, enter the following:

```
: XXXDUMP  ( addr count -- )
   " RAM:XXX" $LOGTO  ( Echo to file RAM:XXX )
   DUMP
   LOGEND
;
```

```
' SWAP  100  XXXDUMP
TYPEFILE RAM:XXX
```

Word Usage Analysis

Using this facility it is possible to keep track what of words are referenced, or not referenced, by your code. This is useful when trimming unneeded words from your program. It can also catch words that were defined and should have been used but weren't.

CLEAR.MARKS (-- , Clears flag in all words to UNUSED state)

START.MARKING.WORDS (-- , Tells FIND to mark all words used.)

STOP.MARKING.WORDS (-- , Turns off this facility)

UNUSED.WORDS (-- , Print all UNUSED words in dictionary.)

USED.WORDS (-- , Print all USED words in dictionary.)

To use this facility, include the utility file "JDEV:UNUSED". Then enter START.MARKING.WORDS . From now on, all words referenced from the keyboard or from a file will be marked as used. You can now load the program that you want to analyze. To find out which words have been referenced by that program, enter USED.WORDS . To find out which words were not referenced, enter UNUSED.WORDS . To clear these flags, enter CLEAR.MARKS .

Error Handling

File: JU:ERROR_CODES

What do you do when something goes wrong? If you can't allocate memory or open a file, what do you do next? It is tempting to simply call ABORT when an error is encountered. This may work fine when debugging an application in Forth because you will be put back into the Forth interpreter. There you can investigate the source of the error and do something about it. When you clone an application, however, abort will cause the program to quit. If your user has just spent 2 hours drawing a masterpiece and the program quits when it can't allocate memory for a brush, you will have failed. An application should always do its best to continue executing and to give the user a chance to save her work.

Although it involves extra work, we recommend that routines subject to errors should return an error flag that is FALSE if everything went OK, and non-zero if an error occured. You can return different codes to indicate different types of errors if more than one is possible. A calling program can check the error code and act appropriately.

We have provided a few tools to simplify, and perhaps standardize, error handling in JForth applications. The first of these is a word that defines succesive error codes.

ERR: (n <name> -- n+1 , define constant)

This was used to define codes for the two most common errors in the following way:

```
1
ERR: ERR_FILE_NOT_FOUND        \ equals 1
ERR: ERR_INSUFFICIENT_MEMORY   \ equals 2
```

```
   DROP  \ don't leave N on stack
```

When you encounter an error, you should inform the user by outputting a text message. You may want to put all of your error message text in a file that is read by your program as needed. This simplifies translation to other languages.

The dreaded GOTO

File: JU:GOTO_ERROR

Many languages have a command called GOTO that lets them jump directly from one place to another. This seems like a handy thing but its use leads to a horrible disease called "spaghetti code". Try to debug some old FORTRAN or BASIC program riddled with GOTOs and you'll know what I mean. It turns out that GOTO is really not needed in a high level language. You can use IF ELSE THEN BEGIN UNTIL etc. to achieve the same effect and produce code that is much more readable and easier to maintain.

You may now all gasp with horror and righteous indignation as I announce the addition of a GOTO to JForth. As I swing by my thumbs over the flames, let me explain why this is really OK.

The one single use of GOTO that has been grudgingly confirmed as a good thing, is when used in error processing. Imagine that you are writing a routine that allocates memory several times. Any one of these allocations could fail. You could use lots of IF statements but soon you will be indenting code right off the side of the screen. An alternative is to use a GOTO.ERROR after each allocation. If the allocation fails, the program will jump to the code following an ERROR: statement. That code can clean up everything and return a proper error code to the caller. Here is a trivial example that demonstrates the use of GOTO.ERROR and ?GOTO.ERROR. It uses local variables because they make it easier to keep the stack clean.

```
include? task-error_codes ju:error_codes
include? task-goto_error ju:goto_error
include? { ju:locals

: TEST.GOTO { aa bb -- error? , simple example }
   aa 0= ?GOTO.ERROR
\
   bb 0<
   IF
      GOTO.ERROR
   THEN
\
   FALSE EXIT  \ exit causes immediate return to caller
\
ERROR:   \ continue from here if error
   >newline ." Uh Oh!" cr
   TRUE
;
```

Try entering:

```
1  2 TEST.GOTO .
0  2 TEST.GOTO .
1 -2 TEST.GOTO .
```

Here is a more complex example that returns different error codes. Instead of calling EXIT we let the code that executes without error continue into the cleanup code.

```
variable BIG-BUFFER \ hold pointers to allocated memory
variable MY-FILE

: DUMP.FILE  ( $filename -- error? )
\
\ try to open file
   $fopen ?dup
   IF
      my-file !
   ELSE
      ERR_FILE_NOT_FOUND GOTO.ERROR
   THEN
\
\ allocate memory to read data into
   memf_clear 200 allocblock ?dup
   IF
      big-buffer !
   ELSE
      ERR_INSUFFICIENT_MEMORY GOTO.ERROR
   THEN
\
\ read file into memory and dump
   MY-FILE @ BIG-BUFFER @ 200 FREAD
   BIG-BUFFER @ SWAP DUMP  \ dump what we got
\
   FALSE  \ return OK
\
ERROR:
\ free memory pointed to by variable if allocated
   BIG-BUFFER FREEVAR
\
\ close file pointed to by variable if open
   MY-FILE FCLOSEVAR
;
```

Caution: when using GOTO.ERROR, make sure that you return the proper items on the stack. Using local variables can greatly simplify this task.

Chapter 12
System Internals

The following memory map shows the layout of the JForth image in memory.

JForth is stored on a disk device in Amiga Binary Format, and when loaded into ram and executed, occupies one contiguous block of memory (with a few specific exeptions, as explained later).

As the image may exist in either position-independent or relocatable form, it may reside at any location in either Chip or Fast ram (uses FAST, if available). For this reason, all addresses mentioned here (and indeed, within JForth itself) are relative to the base of the JForth image, with location 0 being the first (bottommost) byte of the executable program.

USER Variable Data Area

This area holds the actual memory used for storage of USER variables, while the names and executable code used to generate the address of this location reside in the normal dictionary space.

In future versions of JForth, which may support a 'Forth multitasking' environment, each Forth task spawned will receive a local copy of this area. In single-task versions (3.0 and earlier), there is functionally no difference from global VARIABLEs, in fact, the CLONE program will convert USER variables into global VARIABLEs in the standalone image.

The size of the USER area determines the maximum number of USER variables which may be defined; each USER variable taking up 4 bytes. Note that this area may be increased by altering the value stored in #U and subsequently saving the image (via SAVE-FORTH). The saved image, when booted, will have the larger USER area. Once increased, the USER area cannot be made smaller for a given image.

The MAP command will display the number of USER variables defined, as well as the number yet available in the current image

Data Stack Area

Immediately below the USER area is a 32 byte 'buffer-zone', protecting the USER area from programs which underflow the stack by as much as 8 items. No protection is provided for more errant programs.

The Data Stack is assigned just below this zone, starting at the highest location and growing downward as numbers are 'pushed' on to it.

Note that this area is not related to the Return (or 'program') Stack which will be discussed later. This area is solely for the manipulation of data items by Forth operatives such as +, -, etc.

As data is placed on the Data Stack, the Data Stack Pointer (register A6) is decremented, resulting in a data area which grows downward toward the Dictionary Area. Consequently, items removed or

consumed from the Data Stack cause the Data Stack Pointer to approach its original, higher value. Since debugged programs, over the course of running, give back any Data Stack consumed, the normal transient usage of this area does not interfere with dictionary expansion. This is insured by maintaining sufficient Dictionary Area available.

Extensible Dictionary Area

As the JForth vocabulary is extended, new definitions are added serially, consuming the memory immediately above the preceeding one. The act of compilation, then, causes the Dictionary Area to grow upward toward the Data Stack, resulting in an overall lowering of the available Dictionary Space.

As mentioned, the amount of memory needed for the transient Data Stack is relatively small (less than 100 bytes, typically), so Data Stack utilization is not usually a consideration. However, the process of compiling causes the Dictionary to grow toward the Data Stack by an amount directly proportional to the size of the compiled program; without sufficient Dictionary Area available, the Dictionary and Data Stack would meet with undesirable effects.

JForth prevents this by monitoring the distance between the Data Stack and the Dictionary very closely during compiles. The function ?STACK is used for this purpose, aborting the compile if this area becomes less than 256 bytes. Note that you may call ?STACK anytime within your own programs.

Note that the Extensible Dictionary Area may be increased by altering the value stored in #K and subsequently saving the image (via SAVE-FORTH). The saved image, when booted, will have the desired larger Dictionary area.

Each definition added to the Dictionary consumes Dictionary Space; how much is determined by the function type and size. However, regardless of the type and size, each function generates a Dictionary Header made up of a Link Field, Name Field, Size Field and Code Field. The following diagram show the layout of a dictionary header in memory, in order of increasing address:

1. The LINK-FIELD ... 32 bits holding the address (JForth relative) of the preceeding definition's NAME-FIELD. The LINK-FIELD address may be derived from the CODE-FIELD address (see below) with >LINK .

2. The NAME-FIELD ... contains the text of the defined name; the first byte of this field is used as an 'attribute' byte as follows:

The text of the name follows this byte; the text may be followed by a 'filler' byte to insure that the next field is word-aligned. It is the address of the 'attribute' byte that is returned by >NAME , calculated from the CODE-FIELD address (see below).

3. The SIZE-FIELD ... The JForth compiler examines this field to find out how the word is to be compiled (stored in bits 30 & 31) and what the size of the CODE-FIELD is, minus the RTS (stored in the lower word, bits 0-15).

```
D31  D30 === Controls Compilation As Follows:
---  ---

 0    0 .... will be CALLED if size > MAX-INLINE ,
                otherwise INLINE, referred to as "BOTH".
 0    1 .... must be CALLED.
```

```
1    0 .... must be compiled INLINE.

(NOTE: bits 16-29 are reserved for future use).
```

4. The CODE-FIELD ... this field contains the executable 68000 assembly language and completely determines the runtime behavior of the word. It is this address that is returned by ' (tick). The CFA of a word is its *code field address.*

5. The PARAMETER-FIELD ... (optional) In JForth, this refers to the data portion of a CREATE DOES> word ONLY. The word >BODY will convert a CFA to a PFA of such a word. The PFA of a word is its *parameter field address.*

Kernel Dictionary Area

This area is functionally identical to the Extensible Dictionary area in that it is comprised of function definitions which follow the Dictionary Header format described above.

One difference, however is that the source code for this area is not provided in the JForth package, and therefore cannot be recompiled by the programmer (the Kernal is generated by a conventional 68000 Assembler).

The Kernel Dictionary Area is, however, supplied as an assembled package and may be used to regenerate the JForth system for various special cases...

1. The programmer has altered the JForth system files (jf:).
2. Regenerate with a different MAX-INLINE (JForth is normally generated with a MAX-INLINE value of 8).
3. Generate an absolute-minimum system for TURNKEYING (by booting COM:JKERNAL and stating TURNKEYING ON before regenerating).
4. Any other reason the programmer may think of.

For more information on regenerating or customizing the JForth system, refer to the section "How to Generate a New JForth System", later in this chapter.

The lastcompiled word in this area has, for historical reasons, been named TASK .

Other memory Allocation / Utilization...

In the process of using JForth, certain functions will allocate memory for systems use, and do so quite transparently to the user.

Relocations Table

As compiled images approach 140k in overall size, they will begin to assemble JSR operations in the 68000 long absolute mode of addressing.

To the JForth system, this means that a table of information must be kept about these references, so that the Amiga Binary Loader can correctly 'relocate' them when the image is loaded into a different part of memory.

JForth automatically generates and manages this table, completely transparent to the programmer. Note, however, that this table is not maintained within the dictionary; but is allocated from the general memory pool provided by the Amiga.

When present, the ABSOLUTE ADDRESS of the table will be contained in a USER variable called ABSRELOCS .

Each increase of 256 Long Relocations results in an additional 1k of memory allocated from the Amiga OS; 'FORGET'ing programs that contain Relocations will automatically return memory when appropriate.

Files and Memory Housekeeping...

Provided the programmer uses the JForth-provided utilities to allocate memory and open files, JForth will keep track of the requests for these resources, returning them in BYE should the programmer forget.

The data areas used to 'remember' memory and file usage is also allocated outside of the contiguous memory area occupied by the executing JForth image.

JForth Compiler

CFA, is the JForth compiler. It is a defered word set to (CFA,) . The JForth compiler is very flexible. You can define a word to be INLINE , CALLED or BOTH (depending on the system settings when the defined word is compiled). The Forth function + is a BOTH type of word. It is so small, 2 bytes , that it will always be compiled inline, since a call would be at least 4 bytes.

```
DEF +  ( will display )
    ADD.L   (DSP)+,TOS
    RTS
```

Compiling + in another word such as...

```
: USE-OF-PLUS  + + + ;
```

Results in the compiler creating...

```
DEF USE-OF-PLUS
    ADD.L   (DSP)+,TOS
    ADD.L   (DSP)+,TOS
    ADD.L   (DSP)+,TOS
    RTS
```

The default setting to new definitions is CALLED . This is for maximum compatiblity with threaded Forths. So the above USE-OF-PLUS will always be called. Note that USE-OF-PLUS is compiled without any threading of any kind. The code for + is simply laid end-to-end, inline in the dictionary. This eliminates the 36 or more cycles that would be required for a JSR ... RTS between each of the only 10 cycles required for the actual addition. This is one of the main reasons JForth is at least 3 times faster than any threaded Forths. Now look at this word:

```
: EITHER    + + + + BOTH ;
```

The BOTH before the ; tells the compiler to mark this word so that when used by another word, it may be compiled inline or called depending on the value of the MAX-INLINE user variable at compile time. EITHER is 8 bytes long, the RTS is not included in the size. If MAX-INLINE is set to anything less than 8 , a call will be compiled to EITHER . If MAX-INLINE is greater than 8 , the code for EITHER will be laid inline in whatever definition it's used in. So,

```
1 MAX-INLINE !
: EXAMPLE   EITHER  [ 9 MAX-INLINE ! ] EITHER ;
DEF EXAMPLE
    JSR     EITHER
    ADD.L   (DSP)+,TOS
    ADD.L   (DSP)+,TOS
    ADD.L   (DSP)+,TOS
    ADD.L   (DSP)+,TOS
    RTS
```

Note the first EITHER was compiled as a call. After the MAX-INLINE is changed to 9 , EITHER was placed inline. This dynamic sizing feature of JForth allows you to put speed where you need it, and save space where you don't need as much speed.

Some words must be compiled INLINE , like R> >R (LOOP) and others.

How to Generate a New JForth System

These steps guide the programmer in creating a custom JForth development system. This becomes necessary if the programmer has modified any of the 'jf:' (Extras:JForth) or any 'ju:' (JForth:util) files that are part of the normal JForth image.

NOTE: Modifying the JForth System such that this procedure is necessary can seriously affect our ability to render Technical Support, should you request it. You may be asked to furnish a copy of your changes in this event, which can result in delays, if only for shipping purposes. For this reason, we do not recommend modifications in this area unless suggested by Delta Research or otherwise if absolutely necessary.

1) From CLI, type: run com:jkernal

NOTE: number of bytes available will appear in HEX and the I/O will be in SLOW mode. This will change as compilation progresses.

2) When the JForth window appears you can optionally select a tradeoff between speed and memory utilization. If you have memory to burn and want a slightly faster Forth enter:

```
64 MAX-INLINE !
```

For a smaller but somewhat slower version, enter:

```
6 MAX-INLINE !
```

The default is 8 because JForth is plenty fast already and some systems may be short on memory. You can set MAX-INLINE to anything you want but the practical range is 6 to 256. If you set MAX-INLINE too high you may run out of memory. You may want to selectively set MAX-INLINE high just for particular words that you want to optimize.

You are now ready to compile the first stage. Enter:

```
INCLUDE JF:BUILDSYS
```

When the compilation has completed, you need to save this first stage somewhere temporarily. If you have plenty of RAM, you could save it in RAM:MINIMUM. You could also save it on a blank floppy disk or on your hard disk. Assuming you have an area reserved for temporary files called TMP:, enter:

```
SAVE-FORTH TMP:MINIMUM    \ or wherever you want
```

3) Enter: BYE

4) From CLI, enter:

```
RUN   TMP:MINIMUM     \ or whatever you called it
```

5) When the window appears, Enter:

```
INCLUDE   JF:LOADJFORTH
```

The loader will ask if you want to INCLUDE the MODULE facility, the HASHed vocabulary search and History. We recommend answering yes, 'Y', to all these questions. If you do not use Module support then the code will probably not fit in this image. In this case, increase #K then SAVE-FORTH to get a larger image.Modules that are created will be written to the MOD: directory.

6) When the compile has stopped, the programmer may load in any additional files he desires (we recommend adding your files to the list in JF:LOADJFORTH).

7) Type:

```
SAVE-FORTH COM:JForth
```

The just-created COM:JForth image will parallel the released version, reflecting, of course, any changes you have made. You should now re-compile any applications you have written on top of this new JForth image.

Chapter 13
Debugging

by Phil Burk

Debugging is the process of finding out why a program doesn't behave as it should. The bad behavior can be anything from printing the wrong answer to crashing the machine. Finding the source of bugs quickly is a skill that one develops through experience. One can easily spend most of one's precious time debugging. It is worth developing these debugging skills so that one has more time for developing and enjoying code.

Debugging a program really starts before it is written. For information on how to avoid bugs in the first place, please see the sections on programming style.

In this chapter, we will discuss various debugging techniques as well as the various JForth tools specific to debugging.

Tools Overview

These tools can be used together to debug your programs.

DEBUG is JForth's source level debugger. It is described in detail later in this chapter.

DEF will show you the assembly language code that is put together by the JForth compiler. This is not for the faint of heart. It is especially useful for debugging immediate words that compile code, or if you suspect that the compiler is generating code incorrectly. This is extremely rare but can happen. Please call us right away if it does. Please see the section on the Disassembler for more details.

DST will dump an Amiga structure showing you the name of the members and their value. It is in JU:DUMP_STRUCT. Check it out.

DUMP will display the contents of memory at a given address. It is useful for examining data structures like the dictionary, and strings. The stack diagram is:

```
DUMP ( address count -- , dump memory )
```

ECHO (-- addr) is a variable. If you set this to TRUE, the compiler will echo everything as it compiles. This is handy if you are crashing during compilation.

FILE? is handy when you are examining unfamiliar code and need to know how a word is defined. Enter FILE? followed by the name of the word to study.

MAP can sometimes reveal problems related to dictionary size, number of relocations, etc.

TRAPS sets the 68000 exception vectors so that errors like zero divide and odd addressing are trapped. Activating TRAPS will prevent some of the common GURU messages. TRAPS can be turned off with NOTRAPS . TRAPS are automatically installed when you start JForth.

UNRAVEL when executed, displays the return stack as a sequence of subroutine calls. I will pause here to mention that RAVEL and UNRAVEL mean the same thing. Look it up in a dictionary if

you don't believe me. Include JU:UNRAVEL for this. UNRAVEL is useful in a large program when you want to know who is calling a word. Simply put a call to UNRAVEL inside the word that is being called. A typical UNRAVEL display looks like:

```
-150,664
16,576,646
in voc: FORTH in word: (QUIT)
in voc: FORTH in word: INTERPRET
in voc: FORTH in word: DEFER-EXECUTE
in voc: FORTH in word: (INTERPRET)
in voc: FORTH in word: EXECUTE
```

Debugging Hints

Every debugging problem is unique and has a different solution. There is no entirely rational way to find a bug quickly.

Since most debugging is best done intuitively, I have designed this section as a random assortment of hints and questions that may lead you to an answer. At some point in this process you will probably go "Ahah!" and be back in action.

The first step in debugging is to come up with a fairly concise description of the bug. An example might be, "My program is printing one too many numbers." Another might be, "Whenever I click the left mouse button, after drawing a rectangle, I crash.". Once you have a concise description, you can proceed.

ISOLATE THE ERROR. Try to figure out exactly which word is not doing the right thing. Execute each word in the offending area individually to make sure it matches its stack diagram.

WALK THROUGH YOUR CODE. Often a step by step analysis of the offending code is the best way to spot where things go wrong.

CHALLENGE YOUR ASSUMPTIONS. If the code is crashing, obviously something that you assumed was OK is not. Reexamine that code that couldn't possibly be the problem. I have known people to systematically reject all of the code as being the possible source of the error which leads them to the contradictory statement that the bug cannot exist but it does!

DOES YOUR CODE CRASH AT COMPILE TIME? In other words do you get an error while you are INCLUDING a file. If this is true then you might have a syntax error like a missing THEN or a missing ; . You should also check your code for any IMMEDIATE words since they can execute at compile time and cause problems if they are buggy. To find out where the problem is occurring, turn ECHO ON , then include the file. The contents of the file will echo as it is being compiled.

IS THE COMPILER NOT FINDING WORDS THAT YOU THINK IT SHOULD? Check for misspelled words, or a missing (or ; . You might also try REHASH or HASH.OFF in case there is a problem with HASHING. You should also enter ORDER to see what vocabularies are being searched. If you can't get it to recognize any words, try entering:

```
ROOT ONLY FORTH
```

to reset the vocabulary order. Remember that if you are using ANEW, you may have to FORGET a file explicitly in order to INCLUDE something underneath it. If you have just added an INCLUDE? this

might be the problem. You compile the new code then ANEW comes along and FORGETs it! You may also have overwritten part of the dictionary. WORDS will tell you this.

START OVER AGAIN. Sometimes you may be doing things right but are suffering from the lingering effects of a bug that you fixed an hour ago. The old bug may have randomly poked into memory or set some variables wrong. This can be wildly unpredictable. Do a BYE then rerun JForth. You may even want to reboot.

DOES IT WORK THE FIRST TIME THEN NEVER AGAIN? This can be from a number of causes. When you first compile and run, your variables are zero. What are they the second time you run? Did you do an ALLOCBLOCK or an FOPEN at compile time as you include? (Yuck!) If so what happened to that memory or that file? The best way to handle these problems is to have an INIT word that does all required initialization, opens all files and windows needed, allocates all memory needed, initialize all data structures, etc. Then have a TERM word that cleans up all this stuff. Make sure you call the term word before recompiling the code.

DID IT USED TO WORK BEFORE YOU MADE SOME CHANGES? Try to determine exactly what you changed. It is good to test periodically before you go too far along the primrose path. Keep frequent backups.

USE .S LIBERALLY. A few .S calls in your code can be very illuminating. This is not so important now that we have the source level debugger.

ARE YOU OFF BY ONE? Remember DO LOOPs go up to but don't reach their limit. Also remember this old problem. You have a row of boxes labelled 15 through 25 consecutively. How many boxes do you have? Ten? Wrong!! Eleven boxes. Count them. This is known as the old "off by one" problem.

COMMON AMIGA PROBLEMS. Check to make sure that you are passing the Amiga absolute addresses in your function calls and structures. See >ABS and >REL in the glossary. Please remember as well that Forth and the Amiga operating system use a different format for representing strings. Amiga strings are usually passed as NUL terminated 'C' strings. These are a bunch of characters with a 0 at the end. Forth typically have a byte containing a character count at the beginning followed by the characters.

I TYPED SOMETHING BAD AND NOW MY FONT IS MESSED UP! Try entering ^O. That's control-O. This will switch you back to the original font.

Enough random chatter, now let's see how to really debug code.

Source Level Debugger Tutorial

The JForth Source Level Debugger, DEBUG , allows you to single step through your code. This shows you exactly what each word is doing. You can see the result of every DUP , SWAP , ROT , whatever, as it happens.

Compiling with DEBUG{

Let's try out the debugger and see how it works. First we need to load the debugger. It is a good idea to do this before you load the program you are working on. To load the debugger, enter:

```
INCLUDE? DEBUG{ JU:DEBUGGER
```

To debug a word with DEBUG, we need to compile it with the debugger on. You use DEBUG{ and }DEBUG to turn on and off the debugger. When the debugger is on, any word you compile has special information compiled into it for the debugger to use. This makes the code much larger and slower so we only want to do this when debugging. After you finish debugging, make sure you recompile without the debugger on.

Let's define a simple word from the keyboard and then step through it. Enter:

```
DEBUG{   ( turn on debugger )
: SUMN   ( N -- , print N*[N+1]/2 )
         DUP 1+
         * 2/
;
}DEBUG ( turn off debugger )
3 SUMN .
```

After the last line, you should see the answer 6 printed.

Examining Code

We can now single step through this program to see how it arrived at its answer. To debug, we put the word DEBUG right before the word we want to debug. This should be **after** any stack parameters. Enter:

```
0SP  ( clear stack if not already clear )
3 DEBUG SUMN
```

You will see a window called "JForth Debugger" pop up. In the window, you should see a display that tells you you are entering SUMN.

```
Entering:  SUMN >>>>>>>>>>>>>>>>>>>>
------------------------------------------------------------
[1] 3
3DD46 : ------(   DUP                                  |?_
```

You will also see a stack diagram with the 3 on the stack. Below that will be a hexadecimal address and the word DUP. This tells you that DUP is about to execute. The cursor will be at the right of the screen waiting for a key. Press the space bar.

```
<SPACE>
```

You should now see 3 3 on the stack which is what we would expect after DUP finishes. We are now ready to do the 1+ . Keep hitting the space bar until we finish the word. At the end we saw an RTS instruction. This meant that we were about to do a 68000 RTS, Return From Subroutine, instruction. This corresponds to the semicolon at the end of a word.

Now let's add some complexity by calling this word from another. Enter:

```
DEBUG{
: DOSUMS ( -- )
         100 5 DO
                  I DUP . SUMN . CR
         LOOP
;
}DEBUG
DEBUG DOSUMS
```

Keep hitting the space bar until you see SUMN ready to execute. Hit the space bar again. You should now see a 15 on the stack. This is the result of SUMN for N=5. You are now ready to execute the . instruction. Hit <SPACE> and notice that the number 15 appeared in the original program window.

Now let's go through the loop again but this time let's reexamine what SUMN is doing. Keep hitting the space bar until SUMN reappears in the command area. If you miss it just go around again. At this point we have two choices, we can hit space and skim over SUMN like before,or we can dive into SUMN and see it work. With SUMN ready to execute, hit a D key or hit the return key. The display should now say that you are in SUMN. The display of commands will indent to show that you are nested inside another word.

To verify that we are indeed in SUMN, hit the W key. This will display "Where" we are. The display will indicate that DOSUMS has called SUMN.

If you now continue to hit spaces, you will move through SUMN and eventually return to DOSUMS. Thus <SPACE> will advance though a word while <RETURN> will dive into a nested word.

Stopping with a Breakpoint

Now hit the space bar until we get back to the SUMN command. Let's suppose we wanted to move more quickly through the loop. It would be nice if we could just stop before each SUMN, see what N was then zoom forward to the next one. To do this:

```
Hit the B key.
```

You should see a message that a "Breakpoint" was set. This is like putting a stop sign by the call to SUMN. You can have up to 16 breakpoints. To advance quickly:

```
Hit the G key.
```

A 'G' tells DEBUG to "Go" until it hits the Breakpoint or finishes the word. Notice that the answer was printed to the right of the DEBUG display. Hit G a few more times. You can watch the value of N on the stack increasing. You will also see the answers appearing for each time around the loop.

If we get tired of just doing one at a time, we can skip past the breakpoint several times.

```
Hit the # key.
Enter 7 , hit <RETURN>
```

Notice that we saw seven answers appear before it stopped again. If we want to just finish the word, we can "Clear" the breakpoint then "Go".

```
Hit the C key.
Hit the G key.
```

You should now advance through the word at a rapid pace until you finish.

Note: You can also specify a breakpoint by before you run the code using BREAKAT which is described in the Debugger glossary.

Stopping with Control-D

When code is free running and you decide you want to start debugging, you can interrupt the code with a Control-D. Enter

```
DOSUMS  ( let it print a few )
Hit ^D  ( that's Control-D)
```

(Control-D can be hit by HOLDING DOWN the <CTRL> key, then hitting a D, then releasing the <CTRL> key.)

You should now be back in the debugger. Hit W to find out which word you are in.

You can now continue debugging as before.

Debugging a Large Program

If you want to debug a large program from a file, place the DEBUG{ }DEBUG commands around the include statement. For example:

```
DEBUG{
INCLUDE my-file  ( compile one of your programs )
}DEBUG
DEBUG my-word  ( now debug it )
```

If you don't want to have the entire file in DEBUG mode you can place DEBUG{ }DEBUG around individual words in the file. You can also use them within a word since they are defined as IMMEDIATE.

Debugging a Cloned Program

The Debugger will work with Cloned programs. To debug a Cloned program, you must use DEBUG.START and DEBUG.STOP instead of the DEBUG command. Here is an example of a debugging a cloned program.

```
INCLUDE? CLONE CL:TOPFILE
DEBUG{  ( compile with debug )
: TEST  ( -- , clone this puppy )
    DEBUG.START  ( open window and start debugging )
    ." Answer = " 2 2 + . CR  ( fancy program eh? )
    DEBUG.STOP   ( close window )
;
}DEBUG
CLONE TEST
SAVE-IMAGE TEST RAM:TEST

( now in the CLI window, enter: )
RAM:TEST
```

You should see a window open and a normal debugging session will follow. Note: Since Clone removes the Forth headers, you cannot use the 'f' command in the debugger. You may, therefore, want to assign custom functions to keys 7,8,9 using DEBUG.USER.7 , etc. Don't forget to recompile and reclone without the debugger when you are finished. The debugger will add at least 15K to an application and makes it run **much** slower so don't release a program with the debug stuff in there.

IMMEDIATE Words

You will notice that IMMEDIATE words will also show the following word. This is so that words like ' , ..@ , ." , IS , NEW: , etc. will print the word they are operating on. For ." you will not see the entire string, just the first word. Other IMMEDIATE words like IF ELSE and THEN will also show the following words even though this is not so important.

Source Level Debugger Glossary

The words are supported by the Source Level Debugger - DEBUG.

BREAKAT (<word> [<command-in-word>] -- , set breakpoint)

This command can be used to set a breakpoint at a given command in a given word. Consider the following example.

```
DEBUG{
: FOO ( N -- 1 )
   DUP 1+ SWAP - ;     }DEBUG
\  Break  ^ <- right there before SWAP in FOO
BREAKAT FOO SWAP
FOO
```

If you call BREAKAT without a command specified then it will set a breakpoint at the **entry point** of a word. Use NOBREAKS to clear all breakpoints or hit 'C' in the debugger.

DEBUG{ (-- , compile with debug on)

}DEBUG (-- , compile normally)

DEBUG (<word> -- , debug the word whose name follows)

DEBUG.BREAK (-- , enter debugger when encountered)

This word can be sprinkled through your program to force breakpoints.

DEBUG.START (-- , open debugger window and start debugging)

DEBUG.STOP (-- , close window and exit debugger)

DEBUG.USER.7 (-- , deferred action for hitting '7')

You can specify your own function to be executed when a '7' digit key is hit. This allows you to use your own dump and diagnostic routines from the debugger, even when cloned.

```
: MYDUMP    ( -- , dump stuff of interest to user )
    ." my-var = " MY-VAR ?  cr
;
' MYDUMP  IS DEBUG.USER.7
```

DEBUG.USER.8 (-- , deferred like DEBUG.USER.7)

DEBUG.USER.9 (-- , deferred like DEBUG.USER.7)

NOBREAKS (-- , Clear all breakpoints except USER.BREAK?)

USER.BREAK? (address -- break? , user defined break)

This is a deferred word that will be passed the address of the next Forth word to be executed. The word can then decide whether to break into the debugger. This is handy for making logical breakpoints. You could, for instance break if a variable was out of range. Here is an example user break.

```
}DEBUG  ( turn off to avoid recursion )
: MY.BREAK ( address -- break? )
        drop
        VAR1 @ 100 >
;
' MY.BREAK IS USER.BREAK?
```

Debugger One Key Commands

When you are in the debugger,you can hit the ? key for a list of available interactive commands. I recommend playing with these commands to see what they do.

Command Descriptions

Information

```
w - where?, who called who
6 - 680x0 register dump
      Dump all 68000 Data and Address registers.
      See assembler for more information.
m - memory dump from address on stack
      This will treat the top of stack as an address
      and dump the following 32 bytes.
s - regular stack dump
      This just calls .S
r - return stack hex dump
h - here 256 dump
      Accurately display what's at HERE and on the PAD.
      The PAD is at HERE 128 +.
```

Action

```
f - forth, interpret one line
      This will put you in Forth.  You can then
      check variables. Check other words.
      Do whatever.  Enter a blank line to finish.
      The programs stack will be unaffected by your
      actions. Remember, Forth itself is your most
      powerful debugging tool.  (This feature is not
      available in Cloned programs.)
```

```
x - drop one number from stack
n - push a number onto stack
+ - add a number to top of stack
      These last three commands can be used to
      alter the stack contents.
```

Bases

```
1 - decimal, set BASE to 10
2 - binary, set BASE to 2
3 - hex, set BASE to 16
```

User Keys

```
7,8,9 - Execute DEBUG.USER.7,8,9
```

Control

```
b - set the breakpoint here
      Set a breakpoint so that you will stop here
      again if you ever come back.
c - clear the breakpoint
# - enter # breaks to skip
u - up, continue until return
      Finish the current word, go back into
      DEBUG when you return to the calling word.
j - jump over next instruction, don't execute it.
z - set user.break? to 0= , disabled
l - look but don't touch
      The program will continue while displaying
      debugger information.  It will not stop
      for Key Commands until you press a key.
g - go, continue execution without debugging
<SPACE> - single step on same level
<CR> or d - dive down into word
q - quit
      Abort. Handy if continuing will cause a crash.
```

I hope that this debugger will give you a window into your code. It can also be a good learning tool for Forth beginners.

Chapter 14
68000 Assembly

JForth and 68000 Assembly Language

JForth offers an extensive set of tools for operating at the assembly language level. These include:

1. The JForth 68000 Reverse Polish Notation (RPN) Assembler
2. The JForth 68000 Motorola-Style (Forward-Parsing) Assembler
3. The JForth 68000 Disassembler.

This chapter includes documentation for each; some knowledge of 68000 assembly language topics is assumed; for additional technical information on the 68000, refer to a Motorola 680xx Programmers Reference Manual.

JForth Register Utilization

To write successful assembly language programs, a basic understanding of how JForth uses the CPU registers is necessary. This includes which registers are available for general use, and how to push numbers onto the data stack and pop them off.

7 CPU registers are available for general use by the programmer. These include D0, D1, D2, D3, D4, A0 and A1. These may be altered at will, but are not preserved across calls to other JForth words, so relevant registers should be saved and restored if you do so.

JForth requires the remaining 9 registers for itself; these are either unavailable for use by the programmer, or useable only in prescribed ways (such as pushing numbers onto the stack and popping them off).

The following chart describes the JForth CPU register utilization:

Register	JForth Name	Free to Use?	Used in JForth as:
D0	--	Yes	--
D1	--	Yes	--
D2	--	Yes	--
D3	--	Yes	--
D4	--	Yes	--
D5	ILOOP	No	F83 Loop Index #1
D6	JLOOP	No	F83 Loop Index #2
D7	TOS	No	Top of Stack
A0	TEMP0	Yes	--
A1	TEMP1	Yes	--
A2	LOC	No	Pointer to LOCAL stack frame
A3	+64K	No	Pointer to relative addr 64K

```
A4          ORG         No          Pointer to relative addr 0
A5          UP          No          Pointer to base of User Vars
A6          DSP         No          Pointer to 2nd data stack item
A7          RP          No          Return Stack/System Stack
```

Note that JForth provides alternate names for the reserved registers that are more descriptive of their specific functions. These are recognized by both assemblers, and may be optionally included in the output of the disassembler. See each respective section for additional information on register names.

JForth caches the topmost data stack item in the CPU register D7, also named TOS (Top-Of-Stack). The second item on the stack (as well as any others) is stored on an actual stack in memory and is pointed to by register A6, or DSP (Data-Stack-Pointer). Because of this arrangement, pushing a number on the data stack is a two-step process:

1. Move the contents of the TOS register out to the data stack, adjusting DSP value to reflect a new element.
2. Load the new number into TOS.

The process of popping a number off of the stack is even easier, requiring only one step:

1. Move what DSP is pointing to into the TOS register, adjusting DSP value to reflect 1 less element.

Examples of how to do this are given in each assembler section; for now, remember that these operations involve the TOS register (D7) and the DSP register (A6).

One other reserved register can be useful to the programmer, but is ONLY READ FROM, and NEVER CHANGED! This is register A4, also called the ORG register, and contains an absolute pointer to the beginning of the JForth image, the address that JForth considers 0, relative to itself. The value of ORG is often added or subtracted from addresses to convert between the 'real' Amiga ones (ABSOLUTE) and the JForth ones (RELATIVE), as in the words >ABS and >REL.

JForth 68000 Forth Style Assembler (RPN)

Compiling the RPN Assembler

Since the RPN Assembler is available as a MODULE, it is not usually required that the program be compiled. Nonetheless, the Reverse Polish assembler can be compiled into the resident dictionary by entering:

```
DETACHMODULE ASSEM
INCLUDE JF:ASM
```

RPN Assembler Usage

Here is an example of a simple RPN-assembled word.

```
CODE DOUBLE  ( n -- n*2 , Double top of stack.)
    TOS DN  TOS DN  ADD  ( Add TOS to itself.)
END-CODE
7 DOUBLE .
```

The word CODE activates the Reverse-Polish Assembler; subsequent input is interpreted in the ASSEMBLER context. One ADD instruction follows; source operand first, destination operand second, finally the opcode. The whole sequence ends with the END-CODE operator, which deactivates

the ASSEMBLER after installing an RTS instruction as the final 68000 directive in this small program (ALL JForth words MUST end in an RTS). Please note that if an RTS is specifically included as the last instruction in the word being assembled, END-CODE will NOT append another one.

RPN Assembler Register Names

Since they conflict with hex numbers, the normal names for the registers (A0, D4, etc.) cannot be used by the RPN Assembler. Instead, a convention has been adopted whereby the RPN-acceptable name is formed as a single word, beginning with the NUMBER of the register, followed by DR for data registers and AR for address registers. For example, 0AR represents address register 0, 4DR is data register 4, and so on.

Additionally, all of the JForth functional names such as TOS and DSP are available.

Motorola Addressing Modes and RPN Assembler Equivalence

This table describes the various addressing modes available.

```
Forth address mode          Motorola        description
                             example
DN        ( REG ---)          d0       == data register direct
AN        ( REG ---)          a1       == address register direct
A@        ( REG ---)          (a1)     == address register indirect
A@+       ( REG ---)          (a2)+    == A@ then inc reg by size
-A@       ( REG ---)          -(a3)    == dec by size, then A@
AN+W      ( AREG N---)        9(a1)    == address + word indirect
AN+R+B    ( AREG REG BYTE---) 5(a2,d3) == addr+reg+byte indirect
ABS.W     ( N---)             1000     == absolute address word
ABS.L     ( N.LSW N.MSW---)   2000     == absolute address long
PC+W      ( W ---)            7(pc)    == pc + word
PC+R+B    ( REG BYTE---)      9(pc.a2) == pc + reg + byte
#         ( N-OR-D---)        #4       == immediate data
```

Here are some examples of using the different addressing modes with a MOVE instruction, and the Motorola-style equivalent.

```
Source            Destination   Opcode      Motorola equiv.
0DR DN              0AR AN       MOVE    | MOVE.L D0,A0
0AR A@              1AR A@+      MOVE    | MOVE.L (A0),(A1)+
0AR -A@             1AR 50 AN+W  MOVE    | MOVE.L -(A0),50(A1)
1AR 0DR 30 AN+R+B   100 ABS.L    MOVE    | MOVE.L 30(A1,D0.L),100
2000 ABS.W          50 PC+N      MOVE    | MOVE.L 2000,50(PC)
34 #           0DR 100 PC+R+B    MOVE    | MOVE.L #34,100(PC,D0.L)
```

RPN Assembler Support Words.

A data-length may be specified with one of these 3 operators:

```
BYTE -- declares 8 bit data-size
WORD -- declares 16-bit data-size
LONG -- declares 32-bit data-size  ( default )
```

The following examples illustrate the use of the size specifiers:

```
     Reverse Polish                    Motorola
$ 7F #  0DR DN        BYTE AND       AND.B  #$7F,D0
0DR DN  0AR AN        WORD MOVE      MOVE.W D0,A0
0AR 0DR WORD 0 AN+R+B BYTE TST       TST.B  0(A0,D0.W)
```

These two words generate local labels for branching...

```
MARK ( --- )  ( BR# --IN-- )    ( create a label number )
BR:  ( #br -- dest-addr #br  )  ( post fix create a label )
```

The only difference is the manner in which they accept the argument specifying the branch location, either PRE- or POST-FIX notation. This example illustrates both:

```
CODE TEST-SIGN  ( n1 -- result , -1=negative 0=zero 1=positive )
         TOS DN    TST      ( test register contents )
         2 BEQ    ( just leave if its zero by branching to '2' )
         5 BMI              ( if negative, branch to 5 )
         1 #  TOS DN MOVEQ  ( positive if here, set result = 1 )
         2 BRA              ( and exit )
MARK 5   -1 # TOS DN MOVEQ  ( negative if here, set result = -1 )
2 BR:    RTS
END-CODE
```

Note that the numbers used to specify the branch locations need not be sequential, but each one may only specify one location (paired only once with BR: or MARK) within the CODE/END-CODE combination. The same numbers may again be used in the next CODE word.

You may find words in the ASM source code that are not explained in this chapter. These are JForth Assembler INTERNAL words that should NOT be called from other programs.

Non-Standard Opcodes

Those 68000 opcodes that reference the status register (SR), user stack pointer (USP), or condition codes register (CCR) are available in the JForth RPN Assembler as custom opcodes which accept a single source or destination operand. Examples...

ORG TOS 0 AN+R+B MOVE-FROM-SR (move status reg to addr on stack)

$ 0f # ANDI-CCR (zero all but lower 4 bits of CCR)

The entire list follows. IMPORTANT!!! Those marked as "Privileged" should ONLY be used in interrupt code!!!

```
Opcode         Privileged
MOVE-TO-USP          X
MOVE-FROM-USP        X
MOVE-TO-SR           X
MOVE-FROM-SR         X
MOVE-TO-CCR
ANDI-SR              X
EORI-SR              X
ORI-SR               X
ANDI-CCR
EORI-CCR
```

```
ORI-CCR
```

Standard Opcode Mnuemonics

These function according to the Motorola standard, except as noted.

```
LINK MOVEM MOVEP TRAP CMPM MOVEQ
RTR TRAPV RTS RTE RESET NOP STOP SWAP UNLK EXT TAS ABCD SBCD
CLR NEG NOT TST NEGX CMPI ORI ANDI SUBI ADDI EORI MOVE
AND OR SUB ADD EOR CMP MULS MULU CHK DIVS DIVU LEA SUBQ ADDQ
PEA JMP* JSR* NBCD EXG ROR ROL LSR LSL ROXR ROXL ASR ASL ADDX SUBX
Bxx** (branch on condition) DBxx** (decrement & branch on
condition)
BSR***
SCC SLS SCS SLT SEQ SMI SF SNE SGE SPL SGT ST SHI SVC SLE SVS

  * - Use  ] <name> [  instead of Jxx (invokes compiler to CALL
word)
 ** - accepts a local label, for example:   BNE  1$
*** - Usage:   ' <wordname> BSR    (forces PC-relative call)
```

More RPN Examples

```
( Demonstrate accessing a Variable )
VARIABLE MY-VAR
CODE SHIFT-MY-VAR! ( N -- , Shift data in MY-VAR N times left. )
\
\ invoke compiler; generate variable address...
    ] MY-VAR [               ( -- N MY-VAR )
\
\ get a copy of N into d0...
    DSP A@  0DR DN  MOVE     ( -- N MY-VAR ) ( d0=shift count )
\
\ do the shift...
    0DR DN  ORG TOS 0 AN+R+B  ASL   ( -- N MY-VAR )
\
\ drop both numbers from the stack...
    CELL #  DSP AN  ADD       ( -- MY-VAR ) ( same as NIP )
    DSP A@+ TOS DN  MOVE      ( -- )        ( same as DROP )
END-CODE
```

This example demonstrates local branches, and also invoking the compiler (via] and [)to create a call to another Forth word. Note how D0 is saved while calling EMIT.

```
CODE PLOT# ( N -- , emit N dashes )
      TOS DN   0DR DN   MOVE   ( keep count in D0 )
      DSP A@+  TOS DN   MOVE   ( drop count and reload TOS)
1 BR: TOS DN   DSP -A@  MOVE   ( save TOS )
      ASCII - # TOS DN  MOVEQ  ( load '-' character into TOS )
      0DR DN   7AR -A@  MOVE   ( save D0 on return stack)
```

```
        ] EMIT [                ( emit dash )
        7AR A@+   0DR DN  MOVE  ( restore D0 )
        1 # 0DR DN SUBQ         ( decrement loop counter )
        1 BNE                   ( loop until done )
    END-CODE
```

Additional RPN Assembler Features

RPN Assembler Usage in Colon Definitions

As long as you do not use the branching operators (Bcc, DBcc, BR: and MARK), you can activate the RPN assembler during normal compilation, as in the following example. (If you use the assembler as a MODULE and not compiled resident in your dictionary, you need to insure the ASSEM module is loaded via GETMODULE ASSEM for this feature).

```
: SUM*2 ( a b -- [a+b]*2 )
   +  [ ALSO ASSEMBLER  TOS DN TOS DN LONG ADD   PREVIOUS ] ;
```

RPN Assembler Macros

Similarly, to do Macros in JForth RPN assembler, simply write a colon definition that has assembler words compiled into it. (This cannot be done if you need to branch using BR: or MARK. Also, the RPN assembler MUST be resident in the dictionary, and NOT loaded as a MODULE. See the above section *Compiling the RPN Assembler* to achieve that configuration).

```
: M.DOUBLE   [ ALSO ASSEMBLER ]  TOS DN  TOS DN  ADD
             [ PREVIOUS ] ;  IMMEDIATE
```

When this word is used within another definition (as follows), it will invoke the assembler to create the ADD instruction inline.

```
: DOUBLE-MY-VAR ( -- , double the contents of a variable MY-VAR )
  MY-VAR @   ( -- my-var-data )    \ get value in MY-VAR
  M.DOUBLE   ( -- my-var-data*2 ) \ compile above ADD inst inline
  MY-VAR !   ( -- )  ;             \ put it back
```

The RPN assembler is unique in this ability to create macros; this is not currently possible with the Forward Assembler. Unfortunately, the disadvantage of not being able to use the RPN assembler as a MODULE usually outweighs the benfit of this somewhat obscure feature.

The RPN Assembler as a MODULE

In JForth V1.3 and higher, the RPN Assembler may be compiled and saved as a MODULE (as is the default for the release version, in MOD:ASSEM.MOD). The standard procedure for recompiling the com:JForth image, JF:LOADJFORTH, automatically regenerates the .MOD file if MODULE support is included.

When implemented as such, the CODE and ;CODE keywords will automatically load the ASSEM module from MOD: if needed.

When the RPN Assembler is being used in module form, its directives may NOT be used to build assembler macros, described above.

Motorola-Style (Forward-Parsing) Assembler

As many knowledgeable 68000 assembly-level programmers will admit, a non-standard reverse-polish syntax for a Forth assembler seems an additional burden to learn, simply to program in this 'lowest-of-levels'. While the inherent 'macro-ability' and interactivity of the RPN assembler is an important gain, the Forward-Assembler environment features familiar Motorola formats as well as support for 'interactively' accessing elements of the JForth dictionary.

You can see additional examples of Forward Assembler usage in the files JF:HASHING, JIFF:UNPACKING, CL:STARTJFORTH.ASM.

Please note that the Forward Assembler does not currently support the BLOCK environment; it may only be invoked within standard ascii text files accessed with INCLUDE, or from the keyboard.

Compiling the Forward Assembler

Since the Foward Assembler is available as a MODULE, it is not usually required that the program be compiled. Nonetheless, the Forward Assembler may be compiled into the resident dictionary by entering:

```
DETACHMODULE ASSEM
INCLUDE JF:FORWARD-ASM
```

Forward Assembler Usage

The Forward-Assembler is invoked with the "ASM" keyword, followed by the name of the word to be created (similar to the use of ":" when compiling a HIGH-LEVEL word). Assembly-language mnemonics then follow, one statement per line.

Finally, the last line begins (and ends) with the "END-CODE" operator, terminating the assembly-mode and resolving/verifying the just-created word. An 'RTS' instruction is automatically assembled by this operation (ALL JForth words MUST end with RTS!) if necessary.

This is illustrated by the following simple example, called ADD2, which adds the top two items on the stack...

```
ASM ADD2   ( a b -- a+b )
   ADD.L  (A6)+,D7
END-CODE
```

The overall syntax adheres closely with Motorola standards; the following fields, based on character position, are defined:

```
LABEL  OPCODE  OPERAND      COMMENT (rest of line is ignored)
5$:    add.l   (a0)+,d0     add what A0 points to with D0
       move.l  d0,d1        copy the sum into d1
       bne.s   5$           do it again if 'non-zero'
```

The following notes apply to each specific field...

The Forward Assembler Label Field

The first column of each line marks the beginning of the Label Field.

The first character of this field must be one of 3 things: whitespace (blank or TAB), the beginning of a Forth comment, or begin a properly-formatted local label.

The Forward-Assembler supports the use of Motorola-style local labels. A declaration begins in column 1, and consists of the label VALUE followed by "$:" (for example... 18$:).

Currently, only one class of instruction may operate on declared labels; those which conditionally BRANCH on a tested condition. BEQ (branch-if-equal) and DBEQ (decrement-&-branch-until-equal) are examples of this class.

The following illustrates a test & branch condition. Here, the tight loop will be repeated until register D0 equals zero...

```
1$:     subq.l  #1,d0     subtract 1 from d0...
        bne     1$        if not equal to zero, branch to 1$

... <<< EXECUTION CONTINUES HERE WHEN D0 = 0 >>>
```

Please note that label values MUST BE UNIQUE, but only between the same ASM and END-CODE combination. Once another ASM has begun, the values may be reused. Also, any DECIMAL value may be used; the programmer is not restricted to using sequential values.

The Forward Assembler Opcode Field

This field contains standard Motorola-style 68000 mnemonics in either case, and, where necessary, the size-specifying suffix .B, .W, .S, or .L.

A few JForth-specific commands are also scanned for to provide greater flexibility:

If the opcode field contains CALLCFA the next text is considered the name of a JForth word, and a call to that word will be compiled in the most efficient manner possible (do NOT use the JSR or JMP instruction to reference NAMED words...use CALLCFA). The following example reads two variables, DIVIDEND and DIVISOR, calls / to divide them and leaves the result on the stack:

```
ASM DIV2  ( -- n , divide the variables )
        callcfa   DIVIDEND          ( -- DVNDvar )
        move.l    0(org,tos.l),tos  ( -- DVDN )
        callcfa   DIVISOR           ( -- DVDN DVSRvar )
        callcfa   @                 ( -- DVDN DVSR , same as move.l..)
        callcfa   /                 ( -- QUO )
END-CODE
```

It is possible to explicitely force a PC-relative call to be assembled to another named word as long as it is within +-32K of the calling instruction. This is useful to build interrupt code where the ORG register is not yet setup.

```
        move.l    #[ascii -],d0   put a '-' character in d0
        bsr       ProcessChar     call some char processor routine
```

If the opcode field contains FORTH{, all text up to the following } character (or end-of-line) will be sent to the JForth INTERPRETER and executed. This example invokes the compiler to create a reference to an Amiga Library function:

```
        move.l    tos,-(dsp)    ( -- prevTOS )
        move.l    #$40,tos      ( -- $40 )
        FORTH{  ] callvoid dos_lib Delay [  }   ( -- )
```

The Forward Assembler Operand Field

This field describes any operands needed by opcode. All standard Motorola-style effective address formats are recognized, as well as a few JForth-specific patterns which are here described.

The Forward Assembler, as supplied in JForth versions 3.0 and earlier, does not directly support references to the SR (Status Register), CCR (Condition Codes Register) or the USP (User Stack Pointer), but access to the related RPN assembler operatives is always possible. For example...

```
\ Zero all but lower 4 bits of CCR...
FORTH{ ASSEMBLER   $ 0f  #  ANDI-CCR ( andi #$f,ccr )
PREVIOUS }
```

A special construct affords access to most JForth entities; in places where a numeric argument is expected, the [and] characters can be used to delimit a string to be submitted to the JForth INTERPRETer (similar to the FORTH{ operative, described above). This is useful to apply defined data CONSTANTs, but can be used for other purposes. The following example checks if the value in TOS is equal to a pre-defined CONSTANT (in this case, an Amiga mask describing memory):

```
cmp.l  #[MEMF_CHIP],tos     is the value in tos = mask?
```

Another example illustrates how this feature can also be used to pre-calculate operand 'literal' values:

```
move.l #[8 1024 *],tos      put '8K' in TOS
```

A third example reads the 4th element of a 32-bit array, base address in a0, into register d0:

```
move.l [3 cells](a0),d0     read a0 plus 12, indirect
```

Any valid sequence of Forth commands may exist between the [and] characters, as long as it meets the following criteria:

1. The expression has an overall stack diagram of (-- n1) .

2. Due to the parsing requirements of the Motorola assembly format, the text between the [and] delimiters may NOT contain any of the following 6 characters: ( ) , . [ ]

NOTE: You should never use the [and] characters to generate an ADDRESS for an instruction. An example of this would be:

```
move.l  #[' NOOP],tos
```

Such code will run in the JForth dictionary, but is not compatible with the CLONE program. You can use FORTH{ and the compiler directive ALITERAL to create code which will push an address on the stack in a CLONEable way:

```
        FORTH{   ' NOOP   ] ALITERAL [    }
```

NOTE: between the ASM and END-CODE, numbers are INTERPRETed in DECIMAL; preceed hex numbers immediately with a $ character, for example $FFFE.

Example of Accessing Structure Members

Here is an example illustrating how to access members of an Amiga structure. Structure members return their offset when referenced so we can use them directly between [and]. Make sure you use the proper size MOVE. Notice the MOVE.W for the width. Notice also that we clear the high bits of D7 first because MOVE.W onlt sets the low bits. This example also illustrates conversion between absolute and relative addresses. This next word takes a relative window address and prints its title and width.

```
ASM  WINDOW.INFO   ( window -- )
```

```
   MOVE.L   D7,A0       \ relative window address to A0
   ADD.L    ORG,A0      \ convert to absolute address
   CLR.L    D7          \ clear high bits of TOS
   MOVE.W   [WD_WIDTH],D7     ; get width in TOS
   MOVE.L   D7,-(DSP)   \ save TOS
   MOVE.L   [WD_TITLE](A0),D7 ; get title in TOS
   SUB.L    ORG,D7      \ convert to relative for JForth
   CALLCFA  0COUNT      \ ( -- width address count )
   CALLCFA  TYPE
   CALLCFA  SPACE
   CALLCFA  .           \ print width
END-CODE
```

We can test this word by assembling it then entering:

```
INCLUDE JU:AMIGA_GRAPH
GR.INIT  GR.OPENTEST
GR-CURWINDOW @ WINDOW.INFO
GR.CLOSECURW  GR.TERM
```

Example of Referencing Variables from an Interrupt

If you need to reference a variable from an interrupt routine, you cannot use

```
CALLCFA  VAR1  \ NOT legal in interrupt routines
```

because that assumes that the 68000 registers are setup for JForth use. In an interrupt, that will not be true. In an interrupt you cannot use the JForth data stack, or call any word using CALLCFA, or call any word that uses the data stack. Here is an example of referencing a VARIABLE and calling a subroutine using legal PC relative addressing:

```
variable VAR1
ASM INCVAR  \ increment VAR1
   LEA      [VAR1 HERE - 2-](PC),A0
   ADD.L    #1,(A0)
END-CODE
ASM   INTROUTINE
   BSR      INCVAR      \ PC relative call so OK
   CLR.L    D0          \ tell Amiga we're done
END-CODE
```

See the files JD:DEMO_INTERRUPT and JD:HIGH_INTERRUPT for more examples.

The Forward Assembler as a MODULE

In JForth V1.3 and higher, the Forward Assembler may be compiled and saved as a MODULE (as is the default for the release version, in MOD:ASSEM.MOD). JF:LOADJFORTH, the standard procedure for recompiling the com:JForth image, automatically regenerates the .MOD file anew if MODULE support is included.

When implemented as such, the ASM keyword will automatically load the ASSEM module from MOD: if needed.

DISM - JForth Disassembler

Compiling the Disassembler

Since the Disassembler is available as a MODULE, it is not usually required that the program be compiled. Nonetheless, the Disassembler can be compiled into the resident dictionary by entering:

```
DETACHMODULE DISASSEM
INCLUDE JF:DISM
```

Disassembler Output

DISM, and its related forms, sends output to the standard EMIT device. It is of the form:

```
1D14  move.l  tos,-(dsp)       2D07          "-."
1D16  moveq.l #$20,tos         7E20          "~ "
1D18  bsr.w   26D0  = EMIT     6100 09B6     "a..."
1D1C  rts                      4E75          "Nu"
```

The first column indicates the address of the code. This may be displayed in several modes (see DISM, RISM, ADISM and RELDISM below), but normally represents the JForth relative address.

The second column is the assembly language mnemonic.

The third column displays the operands (for options, see DISM-NAMES, below).

In the fourth row is shown the actual hex code of the opcode and related operands.

By default, the fifth column displays ascii-equivalent characters, helpful in identifying strings. (for options, see DISM-CYCLES, below).

"Automatic" Disassembly Features

All forms of the 'automatic' disassembly words will keep track of forward branches, and only stop disassembling when a 'return' type opcode is displayed that does not have a pending conditional branch past it. (?PAUSE is called at the end of each line, though, so you can simply type QUIT to stop disassembling, or space bar to pause and RETURN to continue).

All forms of 'automatic' disassembly will examine two state variables to determine runtime conditions for:

1. whether generic or JForth specific register names will be displayed. (see `DISM-NAMES`, below)
2. whether the timing cycles will be displayed in the 5th column instead of the default ascii information. (see DISM-CYCLES, below)

These state variables are DISM-NAMES and DISM-CYCLES, respectively. For example, to display the timing cycle information, enter DISM-CYCLES ON then disassemble something, observing the 5th column.

Timing cycles are of the form (xx:yy:zz) where:

xx = no. of 68000 bus cycles this instruction

yy = no. of 68000 bus cycles since beginning of word (ignores branches & loops)

zz = no. of microsecs since beginning of word (on std. A2000, 7.16 Mhz 68000)

NOTE: *Timing cycles are estimates at best and largely for entertainment value.*

CPU registers are displayed according to the state of DISM-NAMES as follows:

```
DISM-NAMES = 0    DISM-NAMES != 0
            a0        a0
            a1        a1
            a2        a2
            a3       +64K  image-base+64K
            a4        org  origin
            a5        up   user pointer
            a6        dsp  data stack pointer
            a7        rp   return pointer
            d0        d0
            d1        d1
            d2        d2
            d3        d3
            d4        loc  locals
            d5       iloop 83-type loop index 1
            d6       jloop 83-type loop index 2
            d7        tos  top of stack
```

NOTE: the same flag, DISM-NAMES also determines whether the disassembler will attempt to display the names of routines being 'called'.

Disassembling within the JForth Image

For examining code within the JForth image, the two most often used words are DISM and DEF.

```
DISM  ( rel-addr -- )     \ for example,   ' D+ DISM
```

Disassembles from the relative address on the stack in 'automatic' mode (see above). DISM does not recognize strings, and attempts to disassemble them.

```
DEF   (  <wordname> -- )  \ for example:  DEF D+
```

DEF attempts to find the next input word as a dictionary entry. If it is successful, it disassembles from the code-field-address. Note that DEF will recognize a compiled string, displaying its content in ascii form.

Disassembling outside of the JForth Image

Three words are useful for disassembling code that exists outside of the JForth image.

```
ADISM ( abs-addr -- )
```

ADISM disassembles from the absolute address on the stack, with addresses labeled as such.

```
RISM  ( rel-addr -- )
```

RISM disassembles from the rel-address as does DISM, but displays the first line as address 0, with successive addresses being labeled relative to the beginning of the disassembly.

```
RELDISM ( org-addr rel-addr -- )
```

RELDISM changes the output addressing scheme similar to RISM, but accepts an additional parameter specifying the address to consider relative 0, the origin address.

The Disassembler as a MODULE

In JForth V1.3 and higher, the Disassembler may be compiled and saved as a MODULE (as is the default for the release version, in MOD:DISASSEM.MOD). JF:LOADJFORTH, the standard procedure for recompiling the com:JForth image, automatically regenerates the .MOD file anew if MODULE support is included.

When implemented as such, the following words will automatically load the DISASSEM module from MOD: if needed:

```
DEF  DISM  RELDISM  RISM  ADISM  DISM-DONE?  DISM-WORD?
INIT-DISM  DISM-CYCLES    DISM-NAMES DISM-ORIGIN .REGNAMES?
SHOW-CYCLES
```

When the Disassembler is being used in module form, these words also form the entire programmatic interface to the facility.

Chapter 15
Forth 'BLOCK' Environment

Before the advent of sophisticated operating system environments as that available on the Amiga, Forth systems have been uniquely self-sufficient in their approach to utilizing mass-storage.

The resulting 'BLOCK' environment has been implemented as a pseudo- standard for disk-interfacing, and as such, does provide a small degree of transportability for high-level forth code.

It is important to note, however, that SCREEN files, those created under BLOCK, are not compatible with AmigaDOS text files, being defferent in format (SCREENs do not have end-of-lines).

If you want to convert your BLOCK files to text you can do so using BLOCK2TEXT which is described at the end of this chapter.

AmigaDOS Incompatibilities

Newcomers should note that files produced under the 'BLOCK' environment are not compatible with AmigaDOS text-based programs (editors, word-processors, spoolers, sorters, etc.). For example, they cannot be listed (or printed) via the AmigaDOS TYPE command. For those that wish to use the BLOCK environment, JForth provides both a line editor and a full screen editor, each described below.

Source files are typically a*at least* 3 times larger with BLOCK files, since all white space on a SCREEN is actual ASCII blank characters; there are no end-of-line characters.

The management of source code is often cumbersome in SCREEN files as the programmer is forced to reference sections of the file via SCREEN NUMBERS, sequentially numbered 1024-byte 'blocks' of characters, with no standard means of referencing other files.

For these and other reasons, we strongly discourage the use of SCREEN files for Amiga development work; it is provided for compatibility with existing source code and for those that prefer SCREENS.

JForth will support the standard 'BLOCK' environment, after compiling certain source files; but has been specifically designed to fully utilize the AmigaDOS file-system and accompanying ASCII text files. Certain JForth tools/utilities are not available under the 'BLOCK' environment:

1. TYPEFILE - since SCREEN FILES do not contain end-of-line characters, the output from thisASCII-file oriented program is not readable.
2. READLINE - again rendered unusable, for the same reason.
3. FILE? - not supported for words compiled from SCREENs, it requires end-of-lines.
4. ASM - The Forward Assembler is also line-oriented.
5. DEBUGGER - JForth can only compile in DEBUG mode within ascii files.

JForth supplied SCREEN utilities

Support for the BLOCK environment is provided in 3 files:

1. BLOCK - this file provides the read/write, open/close, and display primitives, along with the necessary buffer management' words.

2. EDITOR - loads a 'line-editor' similar to that promoted by FIG (Forth Interest Group). Defines the EDITOR vocabulary.

3. SCRED - loads the JForth SCReen EDitor that fully interfaces with the above line-editor. Extends the EDITOR vocabulary. Line editor commands may be issued within SCRED.

Issuing the command INCLUDE JDEV:SCRED will cause these files to be compiled; they will require about 25K of dictionary area.

Note that the command INCLUDE JDEV:EDITOR will load only BLOCK and EDITOR, saving about 10K, but providing only line-editing capabilities.

Line Editor Operation and Glossary

To begin an editing session using the line editor, open the screen file via:

```
OPEN-SCR <screen-filename>
```

To enter the EDITOR vocabulary, making available the words listed below, enter:

```
EDITOR
```

To both display SCREEN #1 and select it for editing, type:

```
1 LIST
```

The EDITOR commands may now be used to modify the source. As changes are made, the user may SAVE or SAVE-BUFFERS to write changes to disk, but these changes may not become permanently applied to the file until CLOSE-SCR is executed.

Any time while the SCREEN file is open, you may LOAD the file, specifying the SCREEN number to start LOADing from on the stack.

An example session might be...

```
OPEN-SCR ram:myScreens    \ will ask create? if doesn't exist
EDITOR                    \ expose the EDITOR vocabulary
1 LIST                    \ list out screen 1
1 CLEAR                   \ empty out the entire screen
1 P : Hello  ( -- )       \ enter a small definition
2 P   >newline ." Hello, world!" cr
3 P ;
L                         \ LIST out the entire screen
1 LOAD                    \ LOAD the screen (compile it)
HELLO                     \ execute the word
CLOSE-SCR                 \ all done with the file, tell AmigaDOS
```

The commands in the EDITOR vocabulary...

B (-- , Back-up the cursor by the number of characters at PAD.)

C (-- Copy the following text, inserting it at the cursor.)

```
    C INSERT THIS WHERE THE CURSOR IS
```

D (line# -- , Delete line#, but place its text at PAD.)

E (line# -- , Erase line# with blanks.)

F (<line> -- , search for line that follows)

Search forward from the cursor position, until the text is found, leaving the cursor just past the text. If the text is not found, an error message is printed, and QUIT executed.

```
F THIS STRING
```

H (line# -- , Hold line# at PAD)

Leave it untouched in the SCREEN.

I (line# -- , Insert the line at PAD before line#)

Move line# and all lines below it down. Line 15 is lost.

L (-- , Relist the current screen.)

M (n -- , Move the cursor N characters.)

Negative is to the left. Display the final destination line.

N (-- , Find next occurrence of string last found by F or N.)

P (line# -- , Put new text on a line.)

```
3 P THIS GOES TO LINE 3.
```

R (line# -- , Replace line# with the text in PAD.)

S (line# -- , Spread the screen at line#)

Move it and all lines below it down. Line# becomes blank. Line 15 is lost.

T (line# -- line# , Type line# on the standard EMIT device)

Save its text in PAD.

X (-- , Delete the next occurrence of the following text.)

```
X THIS TEXT
```

CLEAR (scr# -- , CLEAR screen and select for editing.)

COPY (scr1 scr2 -- , COPY screen SCR1 to SCR2.)

DELETE (#chars -- , DELETE #chars before the cursor.)

EMPTY-BUFFERS (-- , Empty current contents of all buffers.)

SAVE-BUFFERS (-- , Write any updated buffers to disk.)

Leave in buffer.

```
SAVE   ( -- , Alias for SAVE-BUFFERS.)

TOP    ( -- , Move the cursor to the top of the screen.)

TILL   ( <line> -- , Delete all text from cursor to located text)
         TILL THIS TEXT

MORE-SCREENS ( #scrs -- , Add #SCRS to current SCREEN file size)
```

SCRED ... the JForth SCReen EDitor

SCRED is a sophisticated editing tool, useful when dealing with SCREEN files. It fully interfaces with the EDITOR line-editor, providing instantaneous access to and from the editing environment.

SCRED incorporates a command line feature, from which any editor or forth command may be executed. This is particularly useful in using line-editor commands while in SCRED.

A SCRED editing session may be invoked with:

```
SCRED <screen-filename>
```

This command will perform the above described line-editor OPEN-SCR, enter the EDITOR vocabulary and display SCREEN 1 of the file.

Once SCRED is entered, the user is presented with a text menu of possible single-stroke key operations. At the top is listed the current screen number and filename while on the left side are the screen line numbers, for use with the line-editor.

To the right is displayed the status of INSERT and WRAP. When SCRED is in INSERT mode, characters typed will 'push' existing characters over, otherwise existing characters are simply overwritten. The INSERT mode may be toggled via a key-stroke listed in the menu.

The WRAP parameter indicates the number of lines that will be affected by the INSERT mode (or when characters are BACKSPACEd). It is set from the command line as follows: 3 WRAP ! or 0 WRAP ! .

```
1 WRAP !
```

The 'Forth Cmd' key will cause the cursor to move below the displayed screen to the 'command line' at the bottom of the window. There it will display the message '<SCRED>:' informing the user that it is waiting for input.

At this point, the user may type in any FORTH or EDITOR command; when the command finishes, the cursor will return to the screen, available for more editing. If the command causes the displayed screen contents to change, its appearance will be updated before the cursor returns.

If the command generates an error, causing the system to execute QUIT, the message 'any key QUITs' will appear in the command line until the user presses a key. This gives him the chance to see any error message in the command line before the SCRED window closes.

SCRED may be exited by typing

```
QUIT
```

or

```
END-ED
```

from the command-line, or simply typing the control key for the EXIT function.

NOTE that *SCRED does not close the SCR-FILE when it exits*. This not only allows the programmer to LOAD the screen(s) he just edited, but also quickly get back to the same SCRED environment by typing:

```
SE
```

This command will reopen the SCRED window to the same screen and cursor position it was last on when SCRED was exited.

Once the programmer has finished with the file and exited SCRED, he should finally close the file with

```
CLOSE-SCR
```

It is this command that tells AmigaDOS to make all the changes permanent on the disk.

BLOCK2TEXT

If you decide you don't want to use BLOCKs, we don't, you can convert your BLOCK file to a normal TEXT file. This will let you use normal text editors like Textra and to use Amiga DOS commands like TYPE. A facility has been provided to do that. It is in the file JDEV:BLOCK2TEXT. To convert a file called PROJECT.blk to a file called PROJEECT.txt, enter:

```
INCLUDE JDEV:BLOCK2TEXT
BLOCK2TEXT PROJECT.blk PROJECT.txt
```

Chapter 16
Precompiled Modules

General Information about Modules

What *are* Modules?

Perhaps the best way to describe the MODULE facility is to describe some of the situations in previous JForth releases it was designed to improve upon...

1. In earlier JForths, all required include files (.j) had to be pre-compiled and resident in the dictionary in order to use them.
2. In earlier JForths, the assembler had to be pre-compiled and resident in the dictionary in order to use it.
3. In earlier JForths, the disassembler had to be pre-compiled and resident in the dictionary in order to use it.

In each, significant dictionary area is used up by definitions that are only occasionally required; this space is also taken in all SAVEd-FORTHs...duplicated in each one.

The MODULE facility allows us to save these sections of dictionary in disk-resident files, and provides a versatile interface to them. Virtually no space is needed in the normal development image, yet at any time, the saved binary module may easily be read and 'hooked in', providing full access to the definitions it contains. Just as easily, the module may later be 'unhooked', restoring the dictionary to its previous state.

When the 'piece of dictionary' is 'hooked in', the VLIST (or WORDS) command will display it's names in keeping with the dictionary location at which they are physically linked in.

Modules are generally only useful with large, 'execute-only' routines (such as the JForth assemblers and disassembler) and data definitions (compiled '.j' header files); these services are implemented as modules in JForth, as released.

The purpose of this document is to describe these services such that the JForth user can quickly benefit from their use. Also discussed are the few restrictions placed on source files destined to be saved as modules, if you wish to create your own.

Modules and SAVE-FORTH

A given module is compatible with all SAVE-FORTHed images that are identical at least from that module definition on down. All the standard JForth modules are automatically created when the system is generated, so as long as you have not changed any JForth system functions, the supplied set of *mod:* files should always work for you.

If you change something in the JForth system, and regenerate *com:JForth*, you simply answer *Yes* to the question dealing with module support when prompted during the rebuild process. All *mod:* files will be regenerated. Your task will then be to recompile your own application-specific JForth programs and

SAVE-FORTH each one. (Alternatively, you could save a copy of the *mod:* directory with each image, then re-assign *mod:* as appropriate).

Technical Notes on Modules

1. The Module service provides a means of allocating a block of memory of a given size, and then shifting subsequent compilation to that area (instead of the normal, JForth-resident dictionary).
2. When the program is finished compiling, compilation is returned to the resident dictionary. However, the linked-list of Forth words now 'jumps' to this non-contiguous area and follows the definitions created by the file. The last of these definitions will point again to the normal dictionary, to the word immediately preceding the word that 'jumped'.
3. Fully compiled, the external memory is then 'unhooked' from the link list and saved to a file. When again read from disk, the reverse process occurs...the existing link-list is broken and the binary module is 'hooked-in'.

The benefits realized are multiple:

* The saved utility is made available in a matter of seconds, a great improvement over the time it would take to compile it.
* The program component does not occupy any contiguous dictionary area, even when loaded.
* Saved images do not contain large, duplicate sections of program, increasing effective disk capacity.
* If the utility is never called, it doesn't use any system resources (memory, dictionary search-time, etc.).

JForth versions 1.3 and 2.0 look for all MODULES in whatever directory you have ASSIGNed the nickname 'mod:' to. "JForth:assigns" defines it as "JForth:modules". In JForth 3.0, additional flexibility is added to the GETMODULE and SEALMODULE functions, described below.

It is important to note that, other than .J structure-type files, module definitions may NOT BE DIRECTLY REFERENCED BY OTHER DEFINITIONS! In other words, they may not be directly called by other programs. The JForth compiler will not compile a reference to a routine outside of the normal dictionary, so you will be notified if you inadvertantly attempt it. Techniques for programmatically accessing module definitions are described ahead (see WILLGET and WILLHIDE).

Using the Assembler and Disassembler Modules

The assembler (JU:FORWARD-ASM, which includes the RPN assembler) and disassembler (JU:DISM) have been provided as pre-compiled modules in the default MOD: directory. Their names are MOD:ASSEM.MOD and MOD:DISASSEM.MOD, respectively.

Both have been implemented transparently such that the module is automatically loaded at the first invocation of any of a specific set of words. For example, the command: DEF SWAP will load the disassembler module if necessary, and disassemble the code for SWAP. The disassembler module will then remain linked in and available until specifically released by the programmer.

The following words will automatically invoke the assembler module:

```
CODE  ;CODE  ASM
```

The following words will automatically invoke the disassembler module:

```
DEF  DISM  RELDISM  RISM  ADISM  DISM-DONE?  DISM-WORD?
INIT-DISM  DISM-CYCLES    DISM-NAMES DISM-ORIGIN .REGNAMES?
SHOW-CYCLES
```

Using other Modules

This section describes the words used to directly manage the loading, usage, and unloading of a given module.

A particular defined MODULE will always be in one of 3 possible states. The first state exists when the JForth image is initially booted (or anytime COLD is executed)...

1. DETACHED... The module has not been read or was purged. No Amiga memory is allocated for it. If the module defines a vocabulary, it is not visible in this state.

   The function: GETMODULE <modulename> enters the next state, when something in the module is needed...

2. LOADED .... GETMODULE <modulename> will read the module from disk into a memory area that has been allocated outside of the JForth image. It will be linked into the dictionary FIND path and its definitions (including its one possible vocabulary) will be accessible.

   When the immediate need for the module is gone, it may be desirable that it not slow down the FIND process until needed again; the module may be 'put away' temporarily with the following command: HIDEMODULE <modulename>. (This is less of a consideration if the HASHed vocabulary search is in effect.)

3. HIDDEN .... HIDEMODULE <modulename> will unlink the modules definitions from the dictionary FIND path, but the module remains in the memory area that has been allocated outside of the JForth image. Its definitions (including its one possible vocabulary) are not accessible; but can be quickly made so with GETMODULE <modulename>, without re-reading from disk.

The programmer, at any time, may return to the DETACHED state, reclaiming all memory used by the module. The DETACHMODULE <modulename> is provided for this purpose.

A fourth (FORTH?) command: .MODULES is provided to display the status of all defined modules. Status is also displayed by the MAP command.

Note that all normal JForth words (FILE?, VLIST etc.) operate normally on module definitions, once GETMODULE has loaded them.

NOTE: The 3.0 versions of GETMODULE and SEALMODULE will accept *one device-type specifier* preceeding the module name (full pathnames are *not* supported). Examples of the legal JForth 3.0 syntax for these commands include:

```
SEALMODULE RAM:MyModule
GETMODULE RAM:MyModule
SEALMODULE devs:MyOtherModule
GETMODULE devs:MyOtherModule
```

Files in INCLUDES Module

The following '.j' files have been compiled into the released version of MOD:INCLUDES.MOD...

```
ji:intuition/intuition.j
ji:intuition/intuitionbase.j
ji:graphics/rastport.j
ji:graphics/text.j
ji:graphics/regions.j
ji:exec/interrupts.j
ji:exec/libraries.j
ji:exec/execbase.j
ji:libraries/dos.j
ji:libraries/dosextens.j
ji:devices/narrator.j
ji:graphics/sprite.j
ji:workbench/workbench.j
ji:workbench/startup.j
```

Creating a Custom Module

The best candidates for new modules are custom structure definitions or larger utilities such as the Assembler or Disassembler. These programs are usually INTERPRETED and EXECUTED, rather than called from other programs. Nonetheless, the ability to support a limited programmatic interface is provided.

Programs that are destined to become modules must conform to the following restrictions:

1. The compiled program may not exceed 32K. This restriction does not apply, however, if the module contains ONLY compiled '.j.' structure-based definitions. Modules of this type may be of any size.
2. No address of any component within the module may be saved within a variable, array, or any other type of storage element. This includes DEFER and GLOBAL-DEFER children or routines which will be used as vectors for such; these may NOT be defined within modules.
3. The use of ' is limited to ONLY those words that are inside the standard JForth dictionary. For example:

   ```
   : NOOP-CFA   ' NOOP  ;
   ```

   is legal within a module, because the word NOOP is in the normal dictionary. However:

   ```
   : NOOP2  ;
   : NOOP2-CFA  ' NOOP2  ;
   ```

   is incorrect since NOOP2 would be resident in the module.
4. The module may define, at most, one VOCABULARY and it must tree from the FORTH vocabulary level.
5. Do not define USER variables within a module...use normal VARIABLEs instead.

The convention in effect for defining the standard modules is to create one source file, which, when INCLUDEd, will create the module definition and .MOD file pair. For example, the file to define a

module for a calculator utility and create the .MOD file would be called MAKECALC and would look something like:

```
\ Create MOD:CALCMOD.MOD from XX:CALC.F (the source file)
\

MODULE CALCMOD   \ define the module in the dictionary

4 MAKEMODULE CALCMOD   \ allocate 4K memory &
                       \ shift compiling to there
                       \ CRASH if not enough mem!

INCLUDE XX:CALCMOD.F   \ compile the source file

cr ." Writing MOD:CALCMOD.MOD" cr cr   \ nice message
SEALMODULE CALCMOD                     \ write the file,
                                       \ will be < 4K
DETACHMODULE CALCMOD                   \ cleanup
```

At this point, if a transparent interface (such as DEF and ASM use) is desired, a WILLGET command must be entered for each word that should automatically load and call the module. The format of this command is:

```
WILLGET <module-name> <function-name>
```

where: <module-name> is the same name as entered for the MODULE and

<function-name> is the name of the word in the module to be executed.

For example, if the main entry point for the above calculator program is called CALC, one more line in the file will be handy...

```
WILLGET CALCMOD CALC
```

will cause the word CALC to automatically load the module, then find and execute the word CALC within it. This is the method for gaining programmatic access to the contents of a module. Note that while even the names of words defined as VARIABLEs, CONSTANTs and STRUCTures may be specified in the WILLGET command, a VOCABULARY name cannot.

A variant of WILLGET is provided which operates identically, except that words defined with it will HIDEMODULE when they finish execution. It's usage...

```
WILLHIDE <module-name> <function-name>
```

For further examples of creating MODULEs, refer to JF:MakeASSEM, JF:MakeDISASSEM and JF:MakeINCLUDES.

If a transparent WILLGET interface to the module is not desired (or reasonable, as in the case of the INCLUDES module), the GETMODULE <modulename> command must be used to gain access to the module definitions.

WARNING: You cannot Clone a program that calls routines in a module in any way, even via a defined WILLGET interface.

Chapter 17
Miscellaneous Development Tools

Command Line History

Since AmigaDOS 1.3, Amiga users have grown accustomed to the convienience of command line history, the feature in the SHELL that allows you to scroll through previous CLI commands using the arrow cursor keys. Since this feature is only present in the CON: device (in AmigaDOS 1.3, the NEWCON: device), and JForth uses RAW: Amiga windows (to aquire single keystrokes for KEY), this feature was not automatically available. We decided, therefore, to add our own command line history that is similar to the Shell history. We also added the capability of loading the function keys with your own custom commands.

Using the Cursor Keys

[Note: Special keys are marked in this text by angle brackets. For example, <CR> , <HELP> , <UP-ARROW> , and <F7> all refer to single keys and not strings.]

The History feature should be present in the normal JForth image. To test this, hit the <HELP> key. You should see a list of function key assignments. If it is not loaded, enter the following:

```
^X  (to clear characters from <HELP> key.
INCLUDE? HISTORY JU:HISTORY
```

The PANIC.BUTTON key <shift-F1> is for use if the Forth Outer Interpreter gets messed up from a program error. If you cannot enter valid commands at the keyboard without getting an error message, try the PANIC.BUTTON. It tries to "simplify" and reset the environment, hopefully giving you a last chance to enter Forth and see what went wrong. History will be automatically turned on when loaded. Now enter the following lines:

```
23 45 67
SWAP .S
DUP . CR
```

Now use the <UP-ARROW> and <DOWN-ARROW> keys to scroll through these previous commands. At any time you can use the <LEFT-ARROW> and <RIGHT-ARROW> keys to move within a line for editing. If you hold the shift key down while hitting <UP-ARROW> you can search back for lines that start with the same string as the current line. For example, enter the following:

```
<CR>
SW<SHIFT+UP-ARROW>
```

You should now see the line that started with SWAP. Shift left and right arrow will move you to the beginning and end of the line respectively.

If you hit <HELP> you will see a list of commands available on the function keys. Notice that some of the commands, like "INCLUDE" have quotes around them. This means that they will be inserted into the input stream. Try these out.

Wouldn't it be nice if you could easily distinguish between the text you entered and what the computer output. You can make your input text be drawn in color 3 by entering:

```
HIGHLIGHT-INPUT ON
CR ." Hello"  ( in a new color )
```

You can use Preferences to change these colors if need be.

Have you ever entered a great Forth word on the keyboard and wished you had done it in a file. You can use LOGTO to save the past keyboard commands into a file. Enter:

```
INCLUDE? LOGTO JU:LOGTO
LOGTO RAM:SAVEIT
HISTORY
LOGEND
```

Now in the CLI enter:

```
TYPE RAM:SAVEIT
```

Note: History will be disabled in a Cloned Application because it is primarily intended as a development tool. Also many Cloned programs use the CLI which already has its own History. If you want to use it in an application, use a RAW: window for your I/O, and remove the definition of KH.EXPECT from CL:REDEFS.F then recompile Clone. The entry in the REDEFS.f file is what disables it when cloned.

How History Works

HISTORY.ON installs a new function into the deferred word EXPECT. This new EXPECT handler performs it's own backspacing, cursor handling, etc. It uses ANSI commands from JU:ANSI to move the cursor and to parse command keys that are hit. As commands are entered they are added a list of commands stored in a byte array. The buffer contains lines with count bytes at each end of each string. This makes it easier to go back and forth in the list.

History Glossary

These commands are part of the JForth History feature.

$>EXPECT ($string -- , command history insert string)

Place a string in the EXPECT buffer. This is used when building function key words that put a string in the input stream. For example, "INCLUDE" which is on <F1> uses this. To add FORGET to <F8> enter the following:

```
: "FORGET"  ( -- , insert FORGET into input )
    " FORGET " $>EXPECT  ( notice space after last letter )
;
' "FORGET" 8 FKEY-VECTORS !
```

See also FKEY-VECTORS .

HIGHLIGHT-INPUT (-- addr , make user input different color)

Set this variable to TRUE to echo your input in color 3. Output from JForth will be in color 1.

HISTORY (-- , print previous commands)

HISTORY# (-- , print history with numbers for XX)

HISTORY.ON (-- , turn on history buffer and function keys)

HISTORY.OFF (-- , turn off history and function keys)

FKEY-VECTORS (key# -- addr , jump table for function keys)

This is an array that holds CFAs for the 10 function keys. Here is an example of loading a function key with a function.

```
: STATE? ( -- , NO stack activity allowed )
    >NEWLINE ." State = " STATE ? CR
;
\  Assign STATE? to function key 7
  ' STATE? 7 FKEY-VECTORS !
\ Now test it by hitting <F7> while compiling.
: FOO ( -- , start compiling )
    ." Hello<F7> Fred" CR ;
<HELP>  ( see it in list )
```

If you want to assign a function to a shifted function key, add 10 to the key#. Thus if you had used 17 instead of 7 in the above example, You could hold down <SHIFT> then hit <F7> to see the value of STATE.

KH_HISTORY_SIZE (-- size , size in bytes of buffer)

If you would like a bigger history buffer, edit the file JU:History and change the constant KH_HISTORY_SIZE to a bigger value. Then regenerate COM:JForth as described in Chapter 12, System Internals, in the section labeled *How to Generate a New JForth System*.

TAB-WIDTH (-- addr , variable containing TAB width)

When a TAB character is entered, a number of BLANK characters are inserted in its place so that the cursor will be at a multiple of TAB-WIDTH.

XX (line# -- , execute previous line)

Use HISTORY# (which, by default, is assigned to a function key) to see a list of previous commands with their numbers. To execute a previous command, just pass that number to XX . Once filled, the oldest command lines are lost from the end of the buffer as new command lines are added.

Vocabularies

Vocabularies provide a way to group words that are to be used in a specific context. Examples within JForth are the ASSEMBLER vocabulary and the DISASSEMBLER vocabulary. Within the ASSEMBLER vocabulary are words like NOT , and SWAP , that are 68000 opcodes. These obviously conflict with existing Forth words. By specifying which vocabulary you are using, one can avoid any ambiguity.

Vocabularies can also be used to speed up compilation by temporarily removing a group of words that don't need to be included in the dictionary search, but this will provide benefit only if the HASHed dictionary search is not in effect. JForth, as released enables the HASHed search.

One can specify the order in which to search a set of vocabularies. JForth uses a CONTEXT stack to determine this order. The system is initialized at cold to:

```
FORTH ROOT
```

This means that JForth will search FORTH, and then ROOT, for any word typed in or encountered while compiling or interpreting. FIND is the word that performs this search. You can display the search order at any time using ORDER . At COLD this will display:

```
Searching (CONTEXT) : FORTH       ROOT
Extending (CURRENT) : FORTH
```

The first line shows the entire search path used by FIND on words that are encountered from the keyboard or a file. These are called the CONTEXT vocabularies, because they define the CONTEXT in which the input stream will be INTERPRETed.

The CURRENT vocabulary is that vocabulary to which new words will be added to as they are compiled, a process also known as 'extending' the vocabulary.

Vocabulary Tutorial

Before we begin exploring vocabularies, lets reset the search order so that we start from a known state. Enter:

```
ONLY ROOT FORTH
ORDER
```

Now let's create a vocabulary called MUSIC that we can place MUSIC related words in. Enter:

```
VOCABULARY MUSIC
ORDER
```

Notice that the MUSIC vocabulary is not automatically placed in the search order. We can add this VOCABULARY to the search order using ALSO. Enter:

```
ALSO MUSIC
ORDER
```

We now would like any music related words we define to be added to the MUSIC vocabulary. For this to happen, the CURRENT vocabulary must be set to MUSIC. DEFINITIONS will copy the top of the CONTEXT stack to CURRENT.

```
: FOO ." Regular old FOO!" CR ;
DEFINITIONS ORDER
```

FOO will have been defined in the normal FORTH dictionary because it was before DEFINITIONS. Everything after will go into the MUSIC vocabulary. Now let's define some music stuff.

```
: NOTES ." fa la la!" CR ;
: SYMPHONY ." da da da duuuuh!" CR ;
: FOO NOTES SYMPHONY ;
FOO
```

The musical FOO is executed because MUSIC is first in the search order. We can get the old FOO by dropping MUSIC from the top of the CONTEXT stack.

```
PREVIOUS DEFINITIONS ORDER
```

```
FOO
```

Vocabulary Glossary

ALSO (-- , dups the top of the vocabulary stack)

Usually this is followed by a vocabulary call as in:

```
ALSO ASSEMBLER
```

This will cause the ASSEMBLER vocabulary to be added to the search order, first in the list.

DEFINITIONS (--)

Sets the current vocabulary to the top vocabulary on the context stack. This results in new definitions being added to whatever vocabulary is first in the search order.

MAXVOCS (-- n , maximum number of vocabularies allowed)

JForth allows 32 different vocabularies to be defined.

ONLY (-- , reinitialize context stack to Forth and ROOT)

If followed by a vocabulary name, the search order becomes that vocabulary, then ROOT, and no others.

ORDER (-- , print CONTEXT stack and CURRENT)

Displays the current search order, left to right.

PREVIOUS (-- , drops the top item off the vocabulary stack)

Removes the first vocabulary in the search order, the one 'just below' becomes firstmost.

SEARCH-CURRENT (-- addr , variable to control search)

FIND checks SEARCH-CURRENT to decide whether to search the current vocabulay after it searches all the context vocabularies. JForth default is to set SEARCH-CURRENT OFF. (It is particularly useful to not search CURRENT when compiling new Forths or Forth-like packages, as a cross-compiler application might do, where execution of a target word would crash the host system).

VOCABULARY (<name> --)

Define a new vocabulary called NAME. When this vocabulary is executed it will place itself at the top of the CONTEXT search order. Vocabularies are not IMMEDIATE.

Vocabulary Internals

VOC-LINK points to the linked list of vocabularies starting with the most recently defined.

Current JForth vocabulary structure:

```
voc-link@:    ( link to other vocabularies )
voc-latest@:  ( point to last word in vocabulary )
```

The structure of vocabularies may be changed in the future. Therefore, use the words we have supplied for moving around the vocabulary structure:

```
VLINK>
VLINK>VLATEST
VLATEST>VLINK
```

VLINK is the linked list address for a particular vocabulary. VLATEST is the address of the vocabulary pointer to the top of its linked list of words. Each vocabulay list of words is terminated with zero in the LINK field.

Here is an example of using these words.

```
VOC-LINK @ .    ( get pointer to last vocabulary )
VOC-LINK @ VLINK>VLATEST @ ID. ( last word in vocab)
VOC-LINK @ VLINK>' >NAME ID.  ( name of vocab )
VOC-LINK @ @ VLINK>' >NAME ID.  ( print next vocab )
```

Look at the VOCS source file if you want to delve deeper.

SHOWHUNKS - for Analyzing Amiga Binary Files

JU:SHOWHUNKS may prove helpful to anyone that needs to get 'into' object files, libraries, executables, any file that conforms to the Amiga Binary Format.

Using the com:JForth image, type:

```
INCLUDE JU:SHOWHUNKS
```

You may be asked about INCLUDEing JU:DISM while this program compiles. If you want to display CODE hunks as disassembled 68000 code (instead of just HEX numbers), instruct the program to use the disassembler.

Also, if there is not enough dictionary space for JU:DISM, you will be informed as such.

When its done compiling, try:

```
SHOWHUNKS C:DIR
```

What you'll be looking at is the CLI 'dir' command file.

CODE_LIMIT is a variable it uses to tell what the largest HUNK_CODE it should display, in cells. Its default value is 100,000 (displays anything less than 100,000 bytes)...which should show you most, if not all, programs you ask for. If you wish to lower this amount to mask out the larger hunks, just enter:

```
100 CODE_LIMIT !  \ only display CODE_HUNKS less than 100 bytes in
size
```

The final interesting thing about this file is that it uses an execution array to perform specific actions on each type of hunk; you may define your own 'array' of vectors (cfa's)... one for each HUNK_TYPE (see the table in the source for examples)...put the base address of the table in ACTIONSARRAY then type:

```
PROCESSHUNKS <FILENAME>
```

See the definition of SHOWHUNKS, last thing in the file.

Yes, I suppose you COULD write your own Loader/Linker with PROCESSHUNKS...

JForth Optimizing Compiler Extensions

This utility extends the JForth Professional V2.x and V3.x compiler such that it will, under certain circumstances, produce smaller, tighter and faster code.

How to use the Optimiser

1. Load the com:JForth image.

2. Compile the file 'jdev:opt.f'.

3. From the keyboard, type: OPT followed by the RETURN key.

At this point the optimizer is installed, and definitions which are loaded on top of this image will be optimized where possible.

Note that the supplied 'jdev:opttest.f' file illustrates the improvement that can be realized. To run this test...

1. Deactivate the optimizer by entering: OPTOFF

2. Run the test by entering: INCLUDE jdev:opttest.f Note the times that are printed to screen

3. Activate the optimizer by entering: OPT

4. Run the test again by entering: INCLUDE jdev:opttest.f Compare with the previous results.

Optimizer Glossary

OPT (--)

Activates the optimizing extensions by setting INTERPRET vector.

NOOPT (--)

De-activates the opimizer, reverts the compiler behavior to normal by restoring the INTERPRET vector.

OPTON (-- , same as OPT)

OPTOFF (-- , same as NOOPT)

Caveats

1. Dictionary size - this version of the optimizer requires about 7-8K of dictionary. Since it would take a very large application to save this much room due to optimizations, the most benefit is felt in CLONEd programs.

2. The keywords OPT and OPTON (same function, different name) will only operate as expected if typed from the keyboard (vs. INTERPRETed from a file).

3. I believe the JU:DEBUGGER utility and OPT do not co-exist well; they tend to 'un-install' each other. If you need to use the debugger, just OPTOFF before starting your debug compile.

4. Don't FORGET the optimizer while it's installed!!!! (IF.FORGOTTEN doesn't help here!)

Technicals (for those who care)

OPT checks for sequential compilation of a subset of JForth words, specifically (in this version)...

```
+       -       DROP    SWAP    DUP     ROT     -ROT
1+      1-      2+      2-      I       C@      C!
@       !       OVER    2DROP   >R      R>      AND
OR      NIP     2DUP    R@      CELL+   CELL-   CELLS
2*      2/
```

LITERAL numbers and CONSTANTS are also optimized.

If OPT finds that, while compiling, any sequence of at least 3 of these words is encountered, it creates code that will, at runtime, cache stack items in the CPU data registers. D1, D2, D3, D4 and D7 are utilized for this purpose (permitting up to 5 stack items to be cached). The registers are not blindly loaded; rather, a "load as needed" algorithm is implemented.

Let's consider the sequence: SWAP ROT +

The normal JForth compiler (w/ MAX-INLINE set to 12) will produce...

```
move.l  (dsp),d0        code for SWAP...
move.l  d7,(dsp)              "
move.l  d0,d7                 "
move.l  $4(dsp),d0      code for ROT...
move.l  (dsp)+,(dsp)          "
move.l  d7,-(dsp)             "
move.l  d0,d7                 "
add.l   (dsp)+,d7       code for +
```

The code produced by the new compiler for the same high-level statements will be...

```
move.l  (dsp)+,d1
move.l  (dsp)+,d2
add.l   d1,d2
```

The first two instructions in the optimized example simply load stack items into the CPU, where the '+' opcode (add.l) processes them.

What happened to the SWAP and ROT operations? Well, the optimizer factors them in at COMPILE time. For example, no code needs to be actually created for a SWAP because the new compiler keeps track of which register currently holds the top item, where any second is, etc. Therefore, after a SWAP, the compiler will simply create opcodes that reference whichever register holding the correct data.

It should be pointed out that the new compiler does not specifically look for the sequence 'SWAP ROT +' to do this... it finds ANY SEQUENCE (of 3 or more) of the above words and puts together optimized code. This means that optimized code will be created if 'SWAP ROT <anything>' is compiled, as long as <anything> is in the above list.

Note that the above code leaves register D2 holding the topmost stack-item and register D7 containing the second one. This is radically different from the normal JForth register usage (D7 holds TOP-OF-STACK, other items in memory on data stack), but as long as we continue to compile words in the above list, the new compiler will account for the non-standard parameter locations.

However, if the new compiler finds that no more optimizations can be done, it will put things back to normal by 'flushing the cache'. For the above example, it would do this by appending the following code...

```
move.l  d7,-(dsp)       push out second item...
move.l  d2,d7           put top item in normal place
```

This code is specific to this example; other combinations will probably leave the top item in other registers (or cache a different number of items) and will require different but similar cleanup code.

PROFILE - Performance Analyser

Ask most programmers and they will tell you that "most programs spend 80% of their time in 20% of their code." If you are going to optimize a program it would be nice to know which 20% to optimize. PROFILE is a performance analyser that will interrupt your code while it is running and keep track of where it is spending its time.

To compile PROFILE, enter:

```
INCLUDE? PROFILE JDEV:PROFILE
```

Now lets write a small program and analyse its performance. Enter in a file and INCLUDE the following:

```
ANEW TASK-TESTPRO

: MOOP    12 0 DO LOOP ;
: BOOP   123 0 DO LOOP ;
: GOOP     1 0 DO LOOP ;

: FOOP
   0 DO MOOP BOOP GOOP
   LOOP
;
```

The words MOOP BOOP and GOOP will take different lengths of time to execute because they loop different numbers of times. Lets pretend we don't know which one is the longest. PROFILE works like MEASURE. It will analyse whatever follows on the command line. Enter:

```
PROFILE  5000 FOOP
```

You will see a short report on how PROFILE is setup then a message that PROFILE has begun. PROFILE is now running your program and interrupting it occasionally. During the interrupt it looks on the return stack to see where the program was executing. To keep track of where this was, PROFILE divides your program into slots. The short report tells you the starting and ending address of the slots and the size of each slot. When an interrupt occurs while your program was in a slot, that slot's counter is incremented. When PROFILE is finished, it analyses the slots and reports on which slots took the most time. It will give you the slot number, its address range, how many interrupts fell into that slot, the percentage of total time, and the percentage of the slots analysed. Check the number of interrupts that you collected to make sure you got enough, at least 10 in the least reported slot. Your program should probably execute for at least a few seconds. On a fast Amiga you may have to do more loops like 20,000 instead of 5,000.

Look at the report from PROFILE. Look at the leftmost column to find the slot number with the highest percentage. Lets assume, for example, that it is 106. To find out what words are in that slot, enter:

```
106 PF.WHO  \ or whatever slot you want
```

Notice that probably all the words in TESTPRO are in that slot. To narrow our analysis down, enter:

```
106 PF.ZOOMIN
```

This will zoom in to just the words in slot 106. Now rerun PROFILE.

```
PROFILE  5000 FOOP
```

Look to see, what slot is now the highest and use PF.WHO to see what words are in there. You will probably find that we are spending most of our time in BOOP which is no surprise.

Remember the first time we ran PROFILE. There was another slot that showed significant activity. Let's zoom back out and rerun PROFILE to see which one that was.

```
PF.ZOOMOUT
PROFILE  5000 FOOP
```

Use PF.ZOOMIN like before, rerun PROFILE. then use PF.WHO to find the culprit. It will probably be ((?DO)) which is a primitive part of a DO LOOP.

PROFILE - Glossary

```
PF.ZOOMIN  ( slot# -- , adjust slots to detail a slot )

PF.ZOOMOUT ( -- , adjust slots to full dictionary )

PF.WHO     ( slot# -- , print what words are in that slot )

PF-INT-RATE ( -- addr , variable holding desired intrs/sec )

PF-%-REPORT ( n -- , threshold percentage to report slot )

PROFILE   ( <executable_forth_statement> -- )
```

PROFILE - Discussion

The technique that PROFILE uses has some good and bad aspects. One problem that can occur is that if your program loops at the same rate as the interrupt occurs, you will always appear to be executing the same small piece of code. To prevent this, try changing the interrupt rate by setting PF-INT-RATE to a different value. Do not go above 1000 or you will bog down the system.

There is another technique for profiling that does not use interrupts. This other technique requires you to compile the program with special profiling code built in. The results can sometimes be more accurate, but program execution can be effected by the extra code. Kernel words that are called in such a system are reflected in the times for the application level words that call them. This may be desirable. A disadvantage of the other system is that execution within a word cannot be analysed. Using PROFILE, a single word can be analysed to see which part is slowest by zooming in.

Chapter 18
Amiga Libraries and Structures

The Amiga is a treasure chest full of wonderful tools: high speed color graphics, pull down menus, speech devices, powerful system libraries. This section of the manual tells you how to use JForth to open that treasure chest. We will describe how to call Amiga Library routines and how to use Amiga structures. We will then describe some of the special tools we have provided to make this easier.

Amiga Libraries - Tutorial

The Amiga system software is organized into libraries that can be used by any program. If you want to open a window, read a file, or draw a rectangle you need to call a library routine to do this. The most often used libraries are DOS, which handles files and processes, GRAPHICS, which handles drawing lines, images, text, etc., INTUITION, which is used to control windows, screens, menus and gadgets, etc., and EXEC which controls the guts of the machine.

Let's start by calling a DOS routine that waits for a given time. Before we call any library routine we must first OPEN that library. Enter:

```
DOS?
```

This will open the DOS library. If it is already open then nothing will happen. (JForth always opens the DOS and EXEC libraries when it starts up but it doesn't hurt to call this again.) Any library that is known to JForth can be opened by calling a similar word, eg. GRAPHICS? or INTUITION? .

We now need to know what arguments, or parameters, this function takes. We should look this up in the DOS Developers Manual to find out everything about this routine. But if you just want to see the argument list you can use ARGS to look them up.

```
ARGS DOS_LIB DELAY
```

ARGS will search through the "FD" files which contain information about the libraries. It prints the argument list and its offset in the library. The offset is handy for assembly language programmers. If the Amiga asks you to insert the disk "FD:" don't worry. It just means that you forgot to execute the ASSIGNS file. This file tells JForth what directory to find other files in. If you get this message, enter in the CLI window:

```
EXECUTE JFORTH:ASSIGNS
```

then hit the "Retry" button on the requestor.

ARGS will tell you that the Delay function takes one argument, the TIMEOUT value. If we look in the manual we will discover that this is expressed in 1/50ths of a second. We will also discover that it does not return a value. Routines that do not return a value are called "VOID". Now that we know what we are calling, let's write a word that calls Delay.

```
: Delay()  ( timeout -- , delay for timeout/50 seconds )
    CALLVOID DOS_LIB DELAY
;
```

Notice that we put two parentheses at the end of the word name. This is a JForth convention that indicates we are calling an Amiga library routine. The word CALLVOID word looks up the information about DELAY just like ARGS did. Then it builds the code necessary to call that routine. We can use the JForth word CALL for routines that do return a value. Now let's test our word.

```
150 DELAY()
```

This should cause a delay of about three seconds. Pretty exciting? Don't worry. Before long you will be opening windows and doing graphics.

Passing Addresses to Library Routines

JForth uses what are called relative addresses. All the Forth words like @ and !, or variables, use an offset from the base of the JForth dictionary. This is so that the addresses will be the same every time you run JForth even though the actual Amiga addresses may be different. This greatly simplifies most programming tasks. If you are passing addresses to Amiga library routines, however, you will need to convert them to absolute 68000 addresses. Luckily, this is very easy to do using >ABS which converts a relative JForth address to a 68000 absolute address. Enter:

```
' DUP .HEX  ( relative address of DUP code )
' DUP >ABS .HEX  ( actual absolute address )
```

Let's call an Amiga routine that requires some absolute addresses. Intuition has a function called CurrentTime that will set two variables to the current time. We will need to pass the absolute addresses of those two variables. Enter:

```
VARIABLE SECONDS
VARIABLE MICROS
: CurrentTime() ( addr-seconds addr-micros -- )
    >ABS SWAP >ABS SWAP  ( convert both addresses )
    CALLVOID INTUITION_LIB CURRENTTIME
;
: PRINT.TIME  ( -- seconds micros )
    SECONDS MICROS CurrentTime()
    ." Seconds = " SECONDS @ .
    ." , Micros = " MICROS @ . CR
;
INTUITION?  ( open library! )
PRINT.TIME  ( test it )
```

Notice how the same word that calls the library routine does the address conversion. That way you only have to pass the normal relative addresses. Whenever possible, try to only pass relative addresses between words.

Now that you know how this works, I will show you a shortcut. If you enter

```
ARGS INTUITION_LIB CURRENTTIME
```

you will see that the arguments are passed in A0 and A1. These are Address registers, as opposed to Data registers like D0 or D1. We can use a special form of CALL that will automatically convert any argument that goes into an address register. This works fine with the GRAPHICS, EXEC, INTUITION and most other libraries. Unfortunately DOS uses data registers to pass addresses (!) so we have to do the >ABS ourselves with DOS.

Here is another way of defining CurrentTime() using this feature:

```
: CurrentTime() ( addr-seconds addr-micros -- )
    CALLVOID>ABS INTUITION_LIB CURRENTTIME   ;
```

This will work even with a mix of address and data arguments. Look at:

```
ARGS GRAPHICS_LIB  DRAW
```

We have been writing simple routines whose only function is to call an Amiga Library routine. You could, of course, use CALL in any colon definition no matter how large. We recommend, however, that you use these small "glue" routines to save space and produce more modular code.

Getting Values from Library Routines

Most Amiga Library routines return a value to the calling program. If, for example, you open a window, the Intuition Library will return to you a pointer to a window structure. You can use this pointer to get information about the window or to perform graphics operations on this window. This pointer will be returned in absolute mode. You should convert this to relative mode before passing it on.

You could use >REL to preform this conversion. A problem can occur, however, if Intuition passes you back a NULL, or zero, pointer. This can happen if Intuition fails to open a window. If you used >REL the NULL would get converted to a nonzero value. You would then be unable to check the pointer to see if it is valid. In this case, therefore, we should use IF>REL for our conversion. It will only convert an address if it is nonzero. Here is how you would call the OpenWindow routine.

```
: OpenWindow() ( newwindow -- window )
    CALL>ABS INTUITION_LIB OPENWINDOW
    IF>REL  ( convert window address )   ;
```

(There is an example of the use of this routine in the JForth Graphics toolbox that we will study later.)

Accessing the Amiga Libraries - Reference

With the installed JForth word set, all the standard Amiga 2.0 libraries are accessible, along with a few more:

```
arp                  asl                  battmem
battclock            clist                commodities
console              cstrings             disk
diskfont             dos exec             expansion
gadtools             graphics             icon
iffparse             input                intuition
keymap               layers               mathffp
mathieeedoubtrans    mathieeesingbas      mathieeesingtrans
mathtrans            misc                 potgo
ramdrive             rexxsyslib           romboot
timer                translator           utility
workbench
```

Opening Libraries

The JForth-recommended method for opening a library is to state the name immediately followed by a question-mark. For example, at the startup of a program which will call both the graphics and intuition

libraries, the programmer need only state 'GRAPHICS?' and 'INTUITION?'. These words will open the library only once, regardless of how many times they are called.

```
: MY-PROGRAM  ( -- )
    GRAPHICS?               \ opens graphics.library if needed
    ...my-graphics-words... \ run application
    -GRAPHICS    ;          \ close the graphics.library
```

JForth opens two of these libraries for you, EXEC and DOS. These libraries are ALWAYS open when the JForth development environment is up. While not an error, it is a null operation to execute either EXEC? , -EXEC, DOS? or -DOS. They are provided only for compatibility reasons and the surrounding JForth environment will actually manage them.

Your program may specify a particular version of a library by storing the desired version number in the user variable called LIBVERSION. Otherwise, JForth will not care which version is found; any will suffice. Note that, if version number IS important to you, it will have to be stated just prior to each library open operation. This is because the variable LIBVERSION is automatically set to zero after each library is opened.

```
36 LIBVERSION !  GRAPHICS?
36 LIBVERSION !  INTUITION?
```

The words that open libraries, such as GRAPHICS?, provide two types of behavior on an error; either to execute QUIT or not (in a CLONEd application, QUIT will exit the program) or not. See the section *Library Open Verification* ahead for details.

Note: When a library is opened, it's **absolute** address is stored in the word called "name_LIB". For example, if you opened the Intuition library, the Intuition library pointer would be stored in INTUITION_LIB . This pointer points to INTUITIONBASE. To get the address of INTUITIONBASE, enter:

```
INTUITION?
INTUITION_LIB @ >REL  ( get relative address of INTUITIONBASE)
```

The same technique can be used to get EXECBASE or other useful library bases.

Closing Libraries.

Closing libraries is similarly easy: library name preceded by a 'minus' sign. The above example application, at termination, can cleanup the two libraries by executing:

```
-GRAPHICS
-INTUITION
```

Note that the '-NAME' words unconditionally close the library, They should, therefore, only be used when an application terminates.

The 'exec' and 'dos' libs cannot be affected by these words. JForth maintains these libraries; the words '-DOS' and '-EXEC' are provided for compatibility reasons and have no code.

Calling Amiga Libraries.

Accessing a library is made easy by JForths 'call compiler'. The format for a system call is as follows:

```
arg1 arg2 ... arg(N)   CALL   libname_LIB  FunctionName
```

```
ARGs = the arguments in the same order as in the Amiga
       technical reference manuals.
'Call'         = invokes the 'call-compiler'
'libname_LIB' = name of library followed by '_LIB'
                 (lower case ok)
FunctionName   = standard, full-text, Amiga name for the
                 function.
```

For example, the dos function 'Seek' is listed in the AmigaDOS Developer's Manual as follows:

```
Seek( file, position, mode )
```

In JForth, a typical SEEK to the beginning of a file might appear:

```
: REWIND.IT  ( -- prev-position , rewind my file )
  ( file     position        mode   )
  MYFILE @       0      OFFSET_BEGINNING call dos_lib seek  ;
```

CALL builds all the necessary code to access the library. It looks up in the "FD:" files to find out which parameters are passed, then calculates the offset of the routine in the library. It then builds the code necessary to pull the parameters off of the stack, place them in the proper 68000 registers and call the routine. CALL may only be used during compilation, for example, inside of a colon definition. It cannot be used interactively from the keyboard.

CALL will normally return the contents of register D0 on the stack. This is the way values are normally returned from the Amiga libraries. Some library functions, however, do not return a value, and some return two. Others even return special values in the Status Register. JForth provides words that instruct CALL to compile slightly different code for these situations.

Library Open Verification

VERIFY-LIBS (-- var-addr , IMPORTANT !!!!!)

If this variable is TRUE (which it is, by default), JForth will compile a check before **every** new library call to make sure that the library is open. This can save you from crashing when you are first debugging a program and might forget to open a library. Once you have your program debugged and are opening the necessary libraries properly, you should set this variable to FALSE and recompile. Since your program will no longer be making these redundant checks, it will run **faster**. Note: CLONE will automatically remove these checks when generating a target image.

```
VERIFY-LIBS OFF   ( then recompile! )
```

LIB_QUIT (-- var-addr , JForth 3.0 and later)

By default, LIB_QUIT is set TRUE. This will cause QUIT to be executed if an error occurs inside a library-opening word such as GRAPHICS?. (Note that within a CLONEd standalone program, QUIT will cause the application to exit).

To prevent this, the program should set LIB_QUIT to false *before <u>every call</u> to a XXX? word which will open a library*. For example...

```
LIB_QUIT off   INTUITION?
LIB_QUIT off   GRAPHICS?
```

To check if the library opened successfully, the program may check the contents of the XXX_LIB variable for a non-zero value. For example...

```
LIB_QUIT off   ICON?
```

```
ICON_LIB @  ( -- lib-pointer / 0 )  \ will be 0 if error occured
```

CALL modifiers

When these words are used before CALL, they will affect the way that the Amiga Library call is compiled. These modifiers only affect the next CALL then they are turned off. In actuality these words are seldom used. We have provided shortcut words that are described in the next section. These modifiers are mainly used if you need several together, for example, RET:DOUBLE and RET:SR.

```
: SAMPLECALL  ( parameters... --  double sr )
    RET:DOUBLE  RET:SR  CALL MATHIEEEDOUBBAS_LIB IEEEDPMul  ;
```

AREGS>ABS (-- , tell CALL to convert addresses with >ABS)

When CALL looks up the parameters in the FD files, it knows which ones are addresses because they get put into address registers. CALL can therefore automatically convert these addresses from JForth relative addressing to Amiga absolute addressing. Warning! DOS passes address in data registers so use >ABS explicitly with DOS calls. NULL addresses are preserved.

RET:DOUBLE (-- , tell CALL to return both D0 and D1)

This is used when you want a 64 bit result. This is used extensively with the double precision floating point libraries.

RET:SR (-- , return Condition Codes from Status Register)

Some of the floating point routines pass back overflow and other flags in the Status Register. This allows you to get those codes. Warning! Do not bypass this facility by calling MOVE-FROM-SR because this instruction is not legal on some 680x0 processors.

RET:VOID (-- , tell CALL not to return a value)

This will instruct the CALL operator to NOT compile the code to put the return value on the stack.

CALL shortcuts

We have provided some special versions of CALL that incorporate these modifiers. These are used more often than using the modifiers directly.

CALL (...parameters... <name_LIB> <function> -- result)

Normal call without modifiers. Any addresses passed should have already been converted using >ABS.

CALL>ABS (...parameters... <name_LIB> <function> -- result)

Just like CALL except parameters destined for address registers are converted automatically. NULL addresses are preserved.

CALLVOID (...parameters... <name_LIB> <function> --)

Don't return anything.

CALLVOID>ABS (...parameters... <name_LIB> <function> --)

Combination of CALL>ABS and CALLVOID .

DCALL (...parameters... <name_LIB> <function> -- double)

The DOUBLE returned occupies 2 stack cells.

As a convention, words that do little more than call an Amiga library routine should have a () suffix placed on them. These Forth words should have the same calling sequence as the Library routine. For example:

```
: DELAY() ( #ticks -- , delay #ticks 1/50ths second)
    CALLVOID DOS_LIB Delay
;
100 DELAY()
```

To find out what parameters a routine expects, you can use the ARGS facility. ARGS will search the '.fd' files and print the line that contains the parameter description.

```
ARGS dos_lib seek   ( prints parameters for seek )
```

Adding Libraries.

The following section applies only if you wish to call a custom library, one not supplied with the original Amiga Workbench release.

The library must be a normal Amiga Library that can be opened using OpenLibrary(). Examples are the ARP library, the Live video digitizer library, Bill Barton's MIDI library, and the AREXX library. (You can also create you own libraries. The details for this are in the Amiga technical literature, and beyond the scope of this discussion.)

Once you've installed your library in the Amiga OS by placing it your LIBS: directory, 'teaching' JForth about it is easy, as illustrated in the following steps, which define a new library called 'GOODIES'. Note that there is NO space between the ':' and the 'L' in this next line.

```
:LIBRARY GOODIES  ( this creates GOODIES_NAME )
                  ( and GOODIES_LIB, used in  )
                  ( the following steps ...   )

: GOODIES?    GOODIES_NAME  GOODIES_LIB  LIB?  ;

: -GOODIES    GOODIES_LIB   -LIB   ;
```

These are the operatives you will use to open your library, in the same manner as that described above under 'Opening Libraries'.

Finally, you will have to construct a 'function declaration' file for your new library that JForth can read and find the calling parameters. This is an ASCII file that can be created with any popular editor and should be placed in the JForth logical volume 'FD:'. Your new file should reside there, and should be similar in format to those which are Amiga-defined; use them as a model, but note the following:

JForth does not require all of the info contained in a standard '.fd' file, so you needn't bother to build the entire text; the only required portions are:

```
##bias xx
Function1(arg1name,arg2name)(arg1reg/arg2reg)
Function2(arg1name,arg2name,arg3name)(arg1reg/arg2reg/arg3reg)
```

The xx above equals the offset from library base for 1st call in list.

For example...the 1st 3 required lines in the DOS_LIB.FD file are:

```
##bias 30
Open(name,access-mode)(D1/D2)
Close(file)(D1)
```

If you were to type that file, you would see that there are more lines present; but these are all that JForth requires.

Once you have compiled the ':Library' statement (along with the 2 opening and closing operators) AND installed a proper '.fd' file (in the logical volume fd:) JForth will allow you to reference that library and its routines by name.

Amiga 'C' Structure Interface

Structures in the Amiga

The Amiga uses "structures" to describe things like windows, screens, icons, fonts, bitmaps, tasks, etc. A structure contains information about these things like width, color, type, etc. All of this information is collected in one area that can be referred to by a single address. Many of the important Amiga routines pass these addresses as a way of referring to windows, menus, etc. Accessing the features of the Amiga requires you to be able to set and retrieve values in these structures. JForth provides tools for accessing these Amiga structures and for defining new ones of your own design.

Loading Structure Definitions from ".j" Files

The data structures that the system uses are typically defined in a set of 'C' include files whose names end in ".h". These files contain templates that describe how the data in a structure is arranged. These files also contain the definitions of named constants. A 'C' program that wants to reference these data structures includes the appropriate ".h" files. For JForth programmers, a set of equivalent files has been created for inclusion in JForth programs. These files have names that end in ".j". They reside in the logical volume "JI:". To load the structures needed for the Serial device, you would enter:

```
INCLUDE JI:DEVICES/SERIAL.J
```

Loading Structure Definitions from Precompiled Modules

Most of the structures that you will need have been precompiled for fast access. They are stored in a module file called MOD:INCLUDES.MOD on the JForth disk. To access these files you must first define the MOD: volume by executing the JForth:ASSIGNS file. Please do this, if you haven't already,by entering in the CLI window:

```
EXECUTE JFORTH:ASSIGNS
```

Now you can enter, in JForth:

```
GETMODULE INCLUDES
```

This will link the structure definitions from the include files to your dictionary. Please see the section on Modules for more information on how this works. You may find that the INCLUDES module does not have a structure that you need. You can then include it as above, add it to the INCLUDES module or make your module as described in the Modules chapter.

Using Structures

Now that the structure definitions are loaded we can make copies from the template. To create one of these structures that you have defined, enter the name of the structure type followed by the name of the new copy. Enter:

```
NewWindow MYNW
MYNW .  ( print address of MYNW for fun )
```

For those familiar with 'C' the first line is equivalent to:

```
STRUCT NewWindow MYNW;
```

This creates a NewWindow structure that is used to describe how you want Intuition to open a new window. To see how this member was defined and the names of its members enter:

```
FILE? NEWWINDOW ( then hit 'y' when prompted )
```

If you want a window that is 300 pixels wide you need to set the WIDTH member of this structure. JForth uses the naming conventions from the Assembler ".i" files so we refer to this member as NW_WIDTH. Enter:

```
300 MYNW S! NW_WIDTH
```

The word S! looks up the offset for NW_WIDTH in the structure and adds it to the address on the stack. It then looks up the size of the NW_WIDTH member and uses the equivalent of ! W! or C! to store the value to that address. By using this syntax we are able to greatly optimize the referencing of structures. (Hackers: Try using S! in a small colon definition then use DEF to examine the code.) We can check that we set the value correctly by entering:

```
MYNW S@ NW_WIDTH .  ( should print 300 )
```

The above two lines in 'C' would be:

```
MYNW.width = 300;
printf ("%d", MYNW.width);
```

Note: In 'C' the case is critical so we cannot use upper case like we normally do for these examples. For actual JForth code we normally use lower case too.

Sometimes you may want to find the size of a structure so that you can allocate memory for it dynamically. To find the size of a structure use the SIZEOF() word or use ALLOCSTRUCT.

```
MEMF_CLEAR
SIZEOF() NEWWINDOW  ALLOCBLOCK  ( allocate structure )
```

Making an Array of Structures

Sometimes it is desirable to have an array of structures. Suppose we want to create 10 gadgets. We could define 10 of them with individual names but it might be more convenient to define an array of gadgets and address them by index. The word ARRAYOF will do that for us.

```
10 ARRAYOF GADGET  MY-GADGETS
3 MY-GADGET . \ print address of gadget #3
```

Referencing Substructures

Sometimes structures contain other structures as members. To access the substructure you will need its address. Once you have its address you can use S! and S@ as before. You can find the address of any member of a structure by using the .. word. An Intuition Screen, for example, contains its own

RastPort. Let us assume we have a variable called SCREEN-PTR that contains the relative address of a Screen. To fetch the value of the foreground pen in its rastport we could use the following:

```
SCREEN-PTR @  ( -- address-of-Screen )
.. SC_RASTPORT ( -- address-of-Screen's-RastPort )
S@ RP_FGPEN . ( print pen value )
```

In 'C' that would be:

```
printf("%d", SCREEN_PTR->RastPort.FgPen);
```

Often structures will contain pointers to other structures. If it is an Amiga structure, then the pointer will be the absolute address of that other structure. A Window, for example, contains a pointer to a RastPort. Assume we have a Window's relative address in the variable WINDOW-PTR. Let's fetch its RastPort's drawing mode.

```
WINDOW-PTR @ ( -- address-of-Window )
S@ WD_RASTPORT ( fetch Rastport's Address
( automatically convert to relative )
S@ RP_DRAWMODE
```

Note that S@ and S! are intelligent about how they handle values. If a structure member is defined as APTR, an address being stored into that member using S! will be converted to absolute using the equivalent of IF>ABS.

Accessing Array Members in Structures

Some structures have members that are arrays of values. A BitMap structure, for example, contains an array of pointers for up to 8 individual bit planes. To reference these arrays, use .. to get the base address of the array, then add the offset required to get to the particular member of the array. To do this you will need to know the width of the values in the array. Suppose we want to fetch the address of the third bit plane in a BitMap called MYBM. We could use the following:

```
MYBM .. BM_PLANES  ( address of 0th bitplane )
2 CELLS +  ( offset to 3rd bit plane ) @
```

Examining Structures with DST

When you are working with structures, it is handy to be able to see all of the values in it at once. JForth provides a tool that makes this easy. Enter:

```
INCLUDE? DST JU:DUMP_STRUCT
```

This is a handy debugging tool. Enter:

```
NEWWINDOW MYNW  ( unless already defined )
120 MYNW S! NW_WIDTH
MYNW DST NEWWINDOW
```

DST will use the structure template whose name follows to dump the contents of a structure whose address is on the stack. The first column is the values of the members. The middle column tells you the width of the member in bytes and whether it is Signed or Unsigned. If a member is signed, then 16 and 8 bit members will be sign extended into a 32 bit number when placed on the stack. Try putting the last line above in a word called DMYNW then experiment with setting members of MYNW then dumping them out.

Defining Your Own Structures

JForth provides tools for you to define your own structure. The syntax is very similar to 'C'. A possible 'C' structure and the corresponding JForth structure are shown below to give you an idea of how they relate.

```
/* A 'C' structure. */
struct datrec   {
    ushort howmany, *sval_ptr;
    long   bigval;
    struct list *alist;
    struct rastport myrastport;
    aptr   some_mem;
    short  table[32];
};
```

In JForth, we should add a prefix to make the member names unique. Let's use "DR_". The same structure in JForth would be defined as:

```
:STRUCT DATREC     ( Start defining a structure )
    USHORT DR_HOWMANY
    APTR   DR_SVAL_PTR ( pointer to a SHORT )
    LONG   DR_BIGVAL
    APTR   DR_ALIST        ( only a pointer )
( member is a complete structure )
    STRUCT RASTPORT DR_MYRASTPORT
    APTR   DR_SOMEMEM
    32 2 * BYTES DR_TABLE    ( make room for array )
;STRUCT     ( Terminate definition )
```

For more examples of how to define your own structures, look at the ".h" files in JI: and compare them to the 'C' includes.

Structure Glossary

The words to support the use of structures are in JU:C_STRUCT and JU:MEMBER. They are loaded as part of the normal JForth image.

Structure Accessing Words

By using S@ and S! instead of ..@ and ..!, you can almost completely ignore relative versus absolute addressing. It will be taken care of for you automatically.

.. (struct-addr <member-name> -- member-addr)

Calculate the address of a structure member by adding an offset to the base of the structure.

..@ (struct-addr <member-name> -- value)

Fetch the value stored in a structure member. This will automatically use the appropriate @ word for that member. Here is a table of the equivalent @ operator used for various member types:

```
BYTE uses C@
SHORT uses W@
```

```
LONG uses @
RPTR uses @
APTR uses @     ( see S@ )
```

The member must be either 1,2 or 4 bytes wide for this to work. For members that are bigger than 4 bytes, eg. an array, use a combination of " .. " and the normal @ and ! words. With this system, you no longer have to worry about the size of your member, just how you use it! If the member is defined as signed BYTE or SHORT then it will be sign extended to 32 bits. This allows you to store negative numbers in 8 or 16 bit members. See B->S in the main glossary for more information about sign extension.

..! (value struct_addr <member-name> --)

Store value into structure member. The word ! , W! , or C! will be used depending on the width of the member as in ..@.

S@ (struct-addr <member-name> -- value)

Equivalent to ..@ except APTR members are converted to relative. The following two lines are thus functionally equivalent:

```
MNW ..@ NW_Title  IF>REL
MNW S@ NW_Title
```

S! (value struct_addr <member-name> --)

Equivalent to ..! except APTR members are converted to absolute. The following two lines are thus functionally equivalent:

```
0" Plots" IF>ABS MNW ..! NW_Title
0" Plots" MNW S@ NW_Title
```

SIGNED-MEMBERS (-- addr , variable to control compilation)

If this flag is TRUE (the default) then when ..@ is compiled it will distinguish between SIGNED and UNSIGNED members. Version 1.2 treated all members as UNSIGNED. If you are having a compatibility problem with 1.2 involving structures, try setting this variable to FALSE and recompile your application.

Structure Defining Words

:STRUCT (<name> -- , Start defining a structure.)

;STRUCT (-- , Terminate a structure definition.)

APTR (<name> -- , Amiga absolute pointer)

This member must be an absolute address when used by the Amiga. If you use S@ and S!, then addresses will be automatically be converted between relative(JForth) and absolute(Amiga) as needed.

ARRAYOF (n <structure> <name> --)

Allocate an array in the dictionary with the given name that has room for N of the specified structures. The name will take an index and return an address just like ARRAY.

```
20 ARRAYOF  GADGET  MY-GADGETS
3  MY-GADGET . \ print address of 3# gadget
```

ALLOCSTRUCT (<structure> -- addr-structure | 0)

Dynamically allocate a structure from memory. This address must eventually be freed using FREEBLOCK.

BYTE (<name> -- , Define a SIGNED 8 bit member)

You do not have to worry about word alignment of subsequent members. SHORT and LONG will automatically place themselves at a word boundary after a BYTE member. This is the way 'C' does it. In assembly you have to put in dummy bytes sometimes to avoid address errors.

BYTES (#bytes <name> -- , define multibyte member)

This is used for defining the other member types. LONG is defined as " 4 BYTES " . Arrays can be put in a structure by multiplying the width of the array units by the number of units. To make a structure member that is 10 LONG words, use:

```
10 4 * BYTES MY_LARRAY
```

LONG (<name> -- , define a 32 bit member)

RPTR (<name> -- , define a relative pointer)

A pointer member that contains a JForth relative address. This will bot be converted by S@ or S!.

SHORT (<name> -- , define a SIGNED 16 bit member)

UBYTE (<name> -- , define an UNSIGNED 8 bit member)

USHORT (<name> -- , define an UNSIGNED 16 bit member)

STRUCT (<struct-type> <name> , structure as member)

Define a structure member that is another type of structure. JForth will look up how big that structure is and make room for the right number of bytes.

Member UNIONs

A 'C' structure sometimes has several members that occupy the same memory space. These members are said to be part of a UNION. These are useful when you want to use the same part of a structure for different purposes. Let's suppose that you are passing a structure that contains a type field and then data that varies with the type. With one type, you want to pass an X,Y pair as SHORT values. With the other type you want to pass a single 32 bit address. You could do this by creating a structure like the following:

```
\ Create flexible structure.
:STRUCT FLEXDAT
    SHORT DATA_TYPE   \ 0 for X,Y, 1 for PTR
    UNION{           ( Start union )
       SHORT XPOS
       SHORT YPOS
    }UNION{
\ Has same offset as xpos, but is 32 bits wide.
       APTR DATA_PTR
    }UNION
    LONG MORE_DATA
```

```
;STRUCT

: REPORT.DATA  ( addr-flexdat -- , Report appropriate data. )
    DUP S@ DATA_TYPE 0 = ( check type )
    IF DUP S@ XPOS ." X = " .
       DUP S@ YPOS ." , Y = " . cr
    ELSE DUP S@ DATA_PTR    ( get pointer to data )
         @ ( get actual data ) ." Data = " . cr
    THEN
    S@ MORE_DATA ." more data = " . cr
;
FLEXDAT FD-1
HEX 01230092 FD-1 ..! DATA_PTR
0 FD-1 ..! DATA_TYPE
REPORT.DATA
1 FD-1 ..! DATA_TYPE
REPORT.DATA
```

In the preceding example, XPOS and YPOS occupy the same position in the structure as DATA_PTR does. If the parts of a union are not the same size, then the size of the largest part will be used. Subsequently defined members will be after that largest part. The following words must always be used in the given order:

UNION{ (-- old-offset new-offset)

Start first half of a union.

}UNION{ (old-offset new-offset -- old-offset max-offset)

Mark next part of union. Reset offset in structure so that subsequently defined members will overlap previously defined members.

}UNION (offset2 --)

Terminates union.

You don't have to worry about these offsets. They are used for communication between the UNION words and can be ignored. Just be careful that you don't put other stuff on the stack that might interfere with these.

Addressing Considerations - Important!!!

JForth uses addresses that are relative to the base of the JForth kernel. This greatly simplifies the usage of JForth because dictionary addresses don't change between successive runs for JForth. One advantage is that you can store JForth dictionary addresses in variables, do a SAVE-FORTH, rerun your code and those addresses are still valid.

The Amiga, however, must use absolute addresses to perform its work. This implies that you may only pass absolute addresses to its library routines. You must also use absolute addresses when setting a pointer in a structure that the Amiga will use. Word like CALL>ABS , S@ and S! help keep track of when these conversions are needed and do them for you. It is rare, therefore, to have to worry about this issue. It is important, however, to understand it so that you can handle unusual situations.

Suppose that you want to place a pointer to a null terminated string inside a NewWindow structure (which, of course, is being prepared for AmigaDOS to process). The word 0" will return a relative address. You must convert this to absolute by using >ABS before placing it in the structure. (S! automatically converts the relative address to absolute; if we use ..! to write the value, we will have to manually convert it using >ABS).

Some Amiga routines use the NULL value for an address to indicate an error, or a special condition. If you use >ABS or >REL on a NULL it is no longer NULL . For these cases you should use IF>REL and IF>ABS to preserve NULL . These are simply defined as:

: IF>REL DUP IF >REL THEN ;

Here is an example of using >ABS and IF>REL

```
INCLUDE? NEWWINDOW.SETUP JU:AMIGA_GRAPH
\ Make an instance of a NewWindow structure.
NEWWINDOW IDEALWINDOW
IDEALWINDOW NEWWINDOW.SETUP  ( Set default values )

: NAME&OPEN ( -- window , Name the window and open it. )
    0" WORK WINDOW"  >ABS   ( convert address )
    IDEALWINDOW ..! NW_TITLE  ( store in structure )
    IDEALWINDOW >ABS    ( Convert for Amiga call )
    CALL INTUITION_LIB OPENWINDOW
\ Convert absolute window address to relative for JForth.
    IF>REL  ( preserve NULL )    ;

: CHECK.WINDOW ( window -- , Check window pointer <> NULL )
    NOT IF ." Window not opened!!" ABORT THEN   ;

NAME&OPEN           \ Get relative address of open window
DUP CHECK.WINDOW
  ..@ WD_RPORT      \ Fetch absolute address of RastPort
  IF>REL            \ convert to relative
  ..@ rp_FgPen .    \ print pen color
```

The absolute address of the RastPort can be passed directly to Amiga graphics routines. If you want to access members of this structure using JForth, you must first convert its address to relative.

Note that in the above example we could have almost entirely avoided having to consider absolute versus relative by using S! and CALL>ABS. I say “almost” because we would still have to convert any address returned by the Amiga using IF>REL just like at the end of NAME&OPEN.

As a general guideline, when passing addresses between JForth words, pass RELATIVE addresses. Do any required conversion to or from absolute, inside your words, before interfacing with the Amiga.

H2J - Convert "xx.h" to "xx.j"

H2J is handy if you want to interface to an Amiga Library that has an associated ".h" file. An example might be the A-Squared Live library or the ARP library. The ".h" file will contain the definitions of

constants and structures to be used with the Library. To use the Library from JForth you will need a ".j" file containing JForth style structure and constant definitions.

When we developed JForth 1.2, we needed something that would convert the Amiga include files. Thus H2J was born.

You can either use the Cloned version of H2J or compile it and use it directly from JForth. To compile H2J, enter:

```
INCLUDE JA:H2J.f
```

This will load ODE and whatever else it needs.

H2J takes two filenames, an input and an output filename. You may use full pathnames.

```
H2J infile outfile
```

To convert newlib.h to JForth style, enter:

```
H2J newlib.h newlib.j
```

H2J will prompt you at various times for one of two things. When it encounters a new structure definition, it will ask you to enter a prefix to add to the member names to make them unique. Like Assembly, Forth requires you to use unique names for the structure members. For a Window structure, for example, we use "wd_". If the structure members already has a prefix, just hit return.

H2J will also ask you to verify it's conversion if it encounters an unusually tricky line. It is usually correct so unless you know it is wrong, just hit return. If it is wrong, type in the line the way it should be.

Don't panic if some little thing goes wrong. You can always go back and edit the file. We find that H2J will convert about 80% of the files completely. The other 20% will require minor tweaking.

Chapter 19
Graphics Toolkit

Overview

These words were designed to give a simple starting point for interfacing to the Amiga graphics library. The words are fairly generic. An application written using them can (and has been) ported to or from other machines with different graphics environments.

If need be, you can also access any of the advanced features of the Amiga by using the CALL and :STRUCT facilities. So while these GR words do not access all of the Amiga facilities, you are not restricted from doing so. By looking at the source code, you will see how to extend this toolkit. You should also look at the Chapter on miscellaneous Amiga tools for more graphics words. For information on using the Amiga libraries, please refer to the *Amiga ROM Kernel Reference Manual*.

The majority of the demos in the JD: directory use this GR system so please refer to them for more examples of its use.

Graphics Tutorial

These words are designed to work with the concept of a current rastport. All drawing commands will act on that rastport. To distinguish these words from other Forth words all of these routines will be prefaced with a 'GR' for GRaphics.

To demonstrate how this system works, let's open a window and do some drawing. The graphics subsystem is not normally loaded in the JForth dictionary. To load this code, enter:

```
INCLUDE? GR.INIT JU:AMIGA_GRAPH
```

The above will load the graphics code from the file AMIGA_GRAPH only if GR.INIT has not already been defined. Before any use is made of the graphics system, we should make sure the graphics system is initialized properly. To do this enter:

```
GR.INIT
```

This sets all of the variables to their proper state and opens the Graphics and Intuition libraries. Now we need to load the information needed to talk to the Amiga libraries. This normally resides in the ".J" include files. These files contain the definitions of structures and constants. Structures are simply a collection of variable organized in a specific way. This organization is defined in the include files. JForth provides the most commonly needed information in precompiled modules. To load the main module, enter:

```
GETMODULE INCLUDES    ( link in precompiled includes )
```

Now we should open a window for drawing. First we need to declare a NewWindow structure. This is a template that gives Intuition some information about the window we desire including its width, height, location, input event flags, etc. This structure will occupy some memory space in the JForth dictionary. We refer to structures by the address of their first byte. Every byte in the computer has an address which is simply a number which uniquely identifies it. When we say the we are "passing a

structure" or "passing a pointer to a structure" we mean that we are passing the address of the structure. (Pointer means address.)

JForth provides a word called NEWWINDOW.SETUP that will set the values in a NewWindow structure to some reasonable values.

```
NEWWINDOW MY-WINDOW        ( define a structure )
MY-WINDOW NEWWINDOW.SETUP  ( set default values in structure)
```

We can use FILE? to see how the NewWindow structure is defined. Enter:

```
FILE?  NEWWINDOW
```

and hit 'Y' when asked if you want to see the source code. Let's examine and change some of the default settings. Enter:

```
NEWWINDOW .   ( print address of first byte just for fun )
NEWWINDOW S@ NW_WIDTH .   ( set by NEWWINDOW.SETUP )
500 NEWWINDOW S! NW_WIDTH   ( let's change it to 500 )
NEWWINDOW S@ NW_WIDTH .   ( did it work )
```

The 500 in the above example was the width of the window in pixels or video dots. We can now pass that structure to the Amiga Intuition Library asking for a window.

```
MY-WINDOW GR.OPENCURW   .   ( open the window )
```

Notice that a window has been opened. You may also have noticed that a value was left on the stack That was the window pointer. You should always check to make sure that this pointer is non-zero. If it is zero, it means that the window did **not** open and your program will probably crash if you pretend that it did.

To draw in this window we can set the current color, and then issue move and draw commands. These commands are based on using an imaginary color pen. To pick up the pen and reposition it you use GR.MOVE . To put the pen down and drag it in a straight line you use GR.DRAW .

```
2 GR.COLOR!
30 30 GR.MOVE
200 55 GR.DRAW    ( draw a line from  30,30 to 200,55 )
300 100 GR.DRAW   ( draw a line from 200,55 to 300,100 )
```

Now let's try drawing a filled rectangle in another color.

```
1 GR.COLOR!
20 20 100 120 GR.RECT    ( draw filled rectangle )
```

We can also output a text message, try entering:

```
3 GR.COLOR!
250 60 " Hello!" GR.XYTEXT
```

The Amiga uses numbers to reference different colors. The Amiga video circuitry displays a picture by reading these numbers from memory and converting them to an actual color. It uses a *color table,* or *pallette*, to figure out what color should be displayed for a given number. These colors can be changed using the Preferences program described in your Amiga user manual.

To finish this session we should close the window and terminate the graphics system.

```
GR.CLOSECURW
```

```
GR.TERM
```

A Simple Graphics Program

Let's experiment with defining a Forth word that draws random vectors. (*Vector* is the graphics industry word for a single line segment.) We can use the Forth word CHOOSE which selects a random value between 0 and whatever value is on the stack. You may want to enter these in a file so that you can modify the program when done.

```
\ Load necessary code.
INCLUDE? GR.INIT JU:AMIGA_GRAPH
INCLUDE? ?CLOSEBOX JU:AMIGA_EVENTS
INCLUDE? CHOOSE JU:RANDOM

ANEW TASK-RANDOM_LINES

NEWWINDOW  MYNW
: DRAW.RAND  ( -- , draw a random vector )
   300 CHOOSE   ( generate random X )
   120 CHOOSE   ( generate random Y )
   GR.DRAW      ( draw line )
;

: MANY.RAND ( -- , draw many random vectors )
   GR.INIT
   MYNW NEWWINDOW.SETUP
   MYNW  GR.OPENCURW
   IF  ( Check to make sure window opened !!!!)
      BEGIN
         DRAW.RAND
         ?CLOSEBOX   ( has closebox been hit )
      UNTIL
      GR.CLOSECURW
   THEN
   GR.TERM
;
```

?CLOSEBOX will return a TRUE if the closebox is ever hit. This gives you a way out of the program.

The word GR.OPENTEST gives you an easy way to open a window for testing. It will automatically ABORT if the window did not open.

Extending the Graphics Toolbox

You may want to add new words that act on the "current RastPort". If so, just remember that the RastPort is stored in ABSOLUTE mode to save us having to convert it before calling the Amiga Libraries. Here is an example of a new routine to draw a single pixel in the current RastPort.

```
: GR.POINT  ( x y -- , draw single point )
    GR-CURRPORT @ -ROT  ( -- rp x y )
    CALLVOID GRAPHICS_LIB WritePixel
```

;

Generic Graphics Glossary

There are three main types of routines in this toolkit. The Control Routines initialize and terminate data structures, and are involved with opening and closing windows. The Output Primitives produce actual graphics output in the current rastport. The Output Attribute involve the appearance, colors, modes, etc. of the Output Primitives.

Control Routines

GR.INIT (-- , Initialize the Graphics Subsystem)

Set attributes to their default values. It also opens the Graphics and Intuition libraries. This word MUST be called before using any of the other words. Calling this routine twice will NOT result in an error.

GR.TERM (-- , Terminate the Graphics Subsystem)

These words are Amiga specific words for setting up NewWindow structures and opening windows.

GR.CLOSECURW (-- , Close CURRENT window if open)

This command looks in the variable GR-CURWINDOW for a window pointer and closes it if one is there. It then clears GR-CURWINDOW and GR-CURRPORT.

GR.CLOSEWINDOW (Window -- , Close an Intuition Window)

If this window is the same as the window stored in the variable GR-CURWINDOW then both GR-CURRPORT and GR-CURWINDOW will be cleared. This is to prevent a window from being closed twice, a fatal error, when working with multiple open windows.

GR-CURWINDOW (-- addr , Variable with rel addr of Window)

GR-CURRPORT (-- addr , Variable with abs addr of RastPort)

This variable contains the ABSOLUTE address of the Rastport. This is used instead of the relative address because most of the Amiga graphics routines require an absolute rastport address. See >REL.

GR.OPENCURW (NewWindow -- Window | 0)

Open a window based on the requested values set in the NewWindow structure. Returns the relative address of a window structure or 0 if it could not open. Be sure to check this return value. If a window does open, this word calls GR.SET.CURWINDOW to make this the current window for drawing.

GR.OPENTEST (-- , Open a window for experimentation.)

This opens a window for testing. It can be used in place of the code in the first example. It uses a NEWWINDOW structure called WINDOWTEMPLATE . This structure can be reused by other programs.

GR.SET.CURWINDOW (Window -- , Sets the current Window)

This sets the current Rastport to this windows Rastport. Subsequent drawing operations, therefore, will take place in this window. If you are using multiple windows, you should save your own window pointers to each of them. You can call this word to determine which of your windows will be drawn into by the GR.xxx routines.

The variables GR-CURRPORT and GR-CURWINDOW are set by this routine.

NEWWINDOW (<name> --INPUT-- , define a NewWindow structure)

NEWWINDOW.SETUP (newwindow -- , set defaults for a new window)

This loads a request structure for a 640 by 200 window that will open in the Workbench screen. It will have a Close gadget and a Sizing gadget. You can modify any of these defaults before calling GR.OPENWINDOW to get different kinds of windows.

Output Primitives

GR.CLEAR (-- , Clear the current drawing surface.)

The rectangle will be based on the current windows drawing surface.

GR.DEHIGHLIGHT (x1 y1 x2 y2 -- , DEHighlight a Rect Region)

Reverse the effect of GR.HIGHLIGHT

GR.DRAW (xpix ypix -- , Draw a line.)

This will draw a line from the current position to xpix, ypix using the current attributes.

GR.FONT! (font -- , calls SetFont() for current RastPort)

See the file JD:DEMO_FONTS for an example.

GR.FONT@ (-- font , get font from current RastPort)

GR.HIGHLIGHT (x1 y1 x2 y2 -- , Highlight a Rectangular Region)

This will highlight a region to bring attention to it. On the Amiga this will be done by XORing with color = 3.

GR.MOVE (xpix ypix -- , Move the current position to xpix, ypix)

GR.NUMBER (n -- , Display number at current position.)

If you need special formatting, you can format the number separately and draw using GR.TYPE .

GR.RECT (x1 y1 x2 y2 -- , Draw a rectangle)

This will draw a rectangle using the current color. The current position will be left at x1, y1.

Warning! Make sure that X2 >= X1 and that Y2 >= Y1. Otherwise the Amiga library routine will overwrite a huge part of chip memory and you will crash. See JD:DEMO_BOXES.

GR.TEXT ($string -- , Draw a text string)

Draw text at the current position. The string must have a byte count at the given address. The current position will be left at the end of the text.

```
" Hello" GR.TEXT
```

GR.TEXTLEN (addr count -- xpixels , x size of string)

Returns length of string in pixels if drawn in current font. This can be used to right justify text by moving XPIXELS to the left of your right margin and drawing from there.

```
100 \ right margin
" Hello" COUNT GR.TEXTLEN -  \ calc x position
50 ( -- x y ) GR.MOVE  \ to start
" Hello" GR.TEXT  \ will end at x=100
```

GR.TYPE (addr count -- , Draw text, like Forth TYPE)

GR.XYTEXT (xpix ypix string -- , Draw a text string)

Draw text at the given position. This is essentially GR.TEXT that does a move first.

Output Attributes

The appearance of these output primitives can be controlled by the setting of output attributes. These attributes remain in effect until changed. The setting words are balanced by query words so that an environment can be saved, changed, and then restored by low level code. The setting words end in ! and the query words end in @ .

GR.COLOR! (color-index -- , Set the color for drawing.)

A color-index is a number that references a color defined in the current pallette. See JD:DEMO_RGB for an instructive example.

```
3 GR.COLOR!
```

GR.BCOLOR! (color-index -- , Set the background color.)

The background color is used when clearing the screen and for filling in around text.

GR.MODE! (mode -- , Set the drawing mode.)

This controls the logic mode that is used to modify the pixels when drawing. You can use the Amiga constants JAM1 , JAM2 , and COMPLEMENT. We have also defined two constants GR_INSERT_MODE and GR_XOR_MODE to promote portability. There is an official bug in the Amiga Library that forces the color to index 3 when in COMPLEMENT mode.

GR.COLOR@ (-- color-index , Fetch the color for drawing.)

GR.BCOLOR@ (-- color-index , Fetch the background color.)

GR.MODE@ (-- mode , Fetch the drawing mode.)

Graphics Input

In event driven systems, all input events should be routed through a top level routine that can handle any event that is generated. These include mouse movement, button presses, menu picks, window close box hits, etc. The type of events generated and the way that they are handled is different in every application. For this reason, we have not included a routine that only handles mouse location input. For information on how to access input events, see the documentation on the EV routines for event processing. You can also examine the DEMO_PAINT file for an example of graphics input.

Event Driven Programming.

A new style of programming is evolving to meet the needs of highly interactive systems. In modern user interfaces, the user is normally free to use any input that the program offers. These might include picking from menus, moving windows around, entering graphic information, hitting the keyboard, or poking at other gadgets on the screen. The program must be ready to respond to any of these input events. A typical program is structured with a loop at the top of the program that gets input events and processes them with a case statement. The loop is exited when, for example, a CLOSEBOX is hit, or QUIT is selected from a menu. Since the user's input events control the flow of the program, this style is referred to as 'Event Driven Programming'.

The Amiga provides support for this style of programming. When a window is opened, the programmer can select which types of events can be generated by setting the IDCMP flags. Messages can then be received from that window that contain one of the selected events or a NULL event if nothing happened. Please refer to the Intuition manual, Chapter 8, that discusses IDCMP and IntuiMessages for details.

Please also examine the files JD:DEMO_MENUS and JD:DEMO_PAINT for examples of how to use this system.

Routines in JU:AMIGA_EVENTS - EV.xxxx

This part of the system normally has to be developed from scratch to meet the needs of individual programs. We have included some simple routines, however, to get people started.

?CLOSEBOX (-- flag)

Checks for a CLOSEBOX hit in GR-CURWINDOW. This is handy if the only kinds of events you want are CLOSEBOX events. All other classes of events are lost. Used in the demos.

EV.2CLICK? (-- flag)

Returns TRUE if last mouse click was the second click of a double click. See JD:DEMO_CLICK

EV.FLUSH (-- , flush all events from queue)

This does EV.GETCLASS with GR-CURWINDOW until a NULL event is received.

EV.GETCLASS (window -- class , get message class or NULL)

This word calls GET.PORT.MSG on the USERPORT associated with the window. If an input event has occurred, the relevant information is extracted and placed in variables for easy access. See PARSE.PORT.MSG below. The message is then replied to using ReplyMsg.

EV.GETXY (-- x y , get last reported mouse position)

This just fetches the values in EV-LAST-MOUSEX and EV-LAST-MOUSEY .

EV.GETXY00 (-- x y , gets x,y corrected for GIMMEZEROZERO)

This can be used if you are getting x,y from a GIMMEZEROZERO window. It corrects for the border of the window.

EV.WAIT (window -- , wait for message, leave in queue)

If you are just waiting for an event, you should call this instead of sitting in a polling loop. Otherwise you will eat up all the CPU time and leave very little for other tasks. When it returns, you can call EV.GETCLASS.

GET.PORT.MSG (port -- class | 0)

This is called by EV.GETCLASS.

PARSE.PORT.MSG (message -- , extract info, place in variables)

The variables are as follows:

```
EV-LAST-CODE
EV-LAST-IADDRESS
EV-LAST-MOUSEX      EV-LAST-MOUSEY
EV-LAST-MICROS      EV-PREV-MICROS
```

This code is in the file JU:AMIGA_EVENTS.

Chapter 20
EZMenu System

Tutorial

The Amiga provides a very sophisticated pull down menu system that greatly enhances the usability of an application. A group of menus can be linked together and associated with a specific window. Each menu has a list of items that the user can select. Each item can be either text or a bitmapped image. Menu picks are obtained by requesting IDCMP messages. When an application receives a menu pick, it executes the appropriate piece of code. A detailed explanation of how this system works can be found in the Intuition Manual.

Using JForth, you can access all of the features of Intuition Menus. Because the Amiga has so many options, however, this can be a somewhat difficult process. We decided to offer a simplified Menu interface that will suffice for most applications. We assume that you would like one or more menus with simple text items. The EZMENU system will build Menu, MenuItem, and IntuiText structures based on this assumption. If you want more fancy menus, you can still use the EZMENU system, but you will need to tweak things a bit.

An example of using the EZMenu system can be found in the file JD:DEMO_MENUS.

EZMenu Structure

The EZMenu system is based on the use of a special JForth structure called an "EZMenu". The EZMenu contains an Intuition Menu structure, a pointer to an array of IntuiText items, a pointer to an array of CFAs (function addresses), and a count of how many items are in the EZMenu. It is defined in the file JU:AMIGA_MENUS . To use the EZMenu system, you first create a number of EZMenu structures. The address of these structures is then passed to the EZMenu routines. By combining the EZMenu routines with the low level Menu routines described later, you can have total control over the menus.

AMENU Program

Let's write a simple menu program that opens a window and attaches to it one menu with three items. We will then scan for Amiga events and process any MENUPICK events. Once you are comfortable with the essentials covered by this program you can explore JD:DEMO_MENUS and JA:EZWALKER.F for more extensive examples.

First we will need to compile the graphics and event support which we studied earlier and the EZMenu code. I recommend typing this program into a file as we go since it is a little long for keyboard entry.

```
INCLUDE? NEWWINDOW.SETUP JU:AMIGA_GRAPH
INCLUDE? EV.GETCLASS     JU:AMIGA_EVENTS
INCLUDE? EZMENU JU:AMIGA_MENUS
```

In the file, after the INCLUDE?s, let's put a call to ANEW. This will help us when we reload this program by automatically forgetting the previous version. We will then create our own EZMenu structure to use with the EZMenu routines. You will need one EZMenu structure for each menu.

```
ANEW TASK-AMENU
EZMENU MY-MENU
```

Now let's write a word that will initialize our menu. We need to allocate room for 3 menu items. EZMENU.ALLOC? will allocate all the needed MenuItem and IntuiText structures needed for the menu. EZMENU.SETUP will link all of these structures together and give the menu a name. The 0 indicates that this is the 0th menu for the menu. Menus and menu items are all numbered starting with zero. We then use EZMENU.TEXT! to give each menu item a name Finally we assign a Command Sequence Key to the "Quit" item. This will allow us to quit by simply holding down the Right Amiga Key and hitting 'Q'.

```
: MY-MENU.INIT  ( -- error? , INITIALIZE MENU )
\ Allocate space for 3 menu items
   3 MY-MENU EZMENU.ALLOC?  \ returns 0 if it fails
   IF
\ Set name of menu and position in list.
      0" Project" 0 MY-MENU EZMENU.SETUP
\ Define the text for each menu item.
      0" Open" 0 MY-MENU EZMENU.TEXT!
      0" Close" 1 MY-MENU EZMENU.TEXT!
      0" Quit"  2 MY-MENU EZMENU.TEXT!
      ASCII Q 2 MY-MENU EZMENU.COMMSEQ!
      FALSE \ flag for NO error
   ELSE
      ." MY-MENU.INIT - Insufficient Memory!" CR
      TRUE  \ error flag
   THEN
;
```

This is all that is required to define the appearance of a menu. If you want to get fancy you could modify some of the menu items to add checkmarks, graphic images, subitems, etc. See EZMENU.ITEM[] for details on how to access individual items.

We now need a word that will process a Menu Pick when it occurs. The Amiga will generate a "menucode" that indicates which item was picked. By using ITEMNUM() , MENUNUM() , and SUBNUM() you can determine exactly which item or subitem in any menu was picked. We only have one menu and no subitems so we only need ITEMNUM().

Let's keep a flag variable that will tell us when to stop. If we hit "Quit" from the menu we can just turn that variable on. If we hit the other two items, just output a message. Nothing fancy. That's for you to add!

```
VARIABLE QUIT-NOW    ( time to stop? )
: DO.WHATEVER ( menucode -- , act on menu item chosen )
    DUP MENUNULL =
    IF DROP   ( not a complete menu pick )
    ELSE ( -- menucode )
        ITEMNUM()  ( -- item# )
```

```
            CASE
                0 OF ." Open File!" CR ENDOF
                1 OF ." Close File!" CR ENDOF
                2 OF ." Quit!" CR QUIT-NOW ON ENDOF
            ENDCASE
        THEN
    ;
```

The program will probably be receiving more than just Menu Picks. There will be Mouse Clicks, CloseWIndow events, etc. We need to treat the differently. This next word passes Menu Picks to DO.WHATEVER and also turns on the quit flag if the CloseWindow Gadget is hit. The information on what menu item was picked is stored in EV-LAST-CODE by the EV.GETCLASS word. This word is described in the section on Event Handling.

```
    \ Process IDCMP events.
    : HANDLE.EVENT ( eventclass -- )
        CASE
    \ Perform Menu actions
            MENUPICK
            OF  EV-LAST-CODE @ DO.WHATEVER
            ENDOF
    \ Set quit flag if CLOSEBOX hit.
            CLOSEWINDOW
            OF QUIT-NOW ON
            ENDOF
        ENDCASE
    ;
```

This last word only handles one event. We, therefore, need a loop that will scan for events in our application window and pass them to HANDLE.EVENT for processing. We use GR-CURWINDOW, which holds a pointer to the current window, to get events from. The loop will quit when the QUIT-NOW flag is turned on.

```
    : LOOP.MENU ( -- , poll for events )
       QUIT-NOW OFF
       BEGIN
    \ wait for an event so we don't tie up the Amiga
          GR-CURWINDOW @ EV.WAIT
    \ Check for events in the current window.
          GR-CURWINDOW @ EV.GETCLASS ?DUP
          IF HANDLE.EVENT
          THEN
          QUIT-NOW @
       UNTIL
    ;
```

Now we just need to tie all this together. We start by initializing graphics and opening a window. I then initialize the menu and attach it to our window using SetMenuStrip(). When I terminate I detach the menu with ClearMenuStrip() and also free the memory declared by EZMENU.ALLOC. I then run the whole thing from the AMENU word. I have found that this style of organizing a program, with a

clear INIT and TERM helps reduce bugs and makes for more readable code. Note how the error codes from GR.OPENCURW and MY-MENU.INIT propagate up to the highest level.

```
NEWWINDOW MYNW
: AMENU.INIT ( -- error? , set everything up )
   TRUE \ set default error return
   GR.INIT
   MYNW NEWWINDOW.SETUP
   MYNW GR.OPENCURW  \ returns &window if successful
   IF
\ Initialize menu and attach to window.
      MY-MENU.INIT 0=
      IF
         GR-CURWINDOW @ MY-MENU SETMENUSTRIP()
         DROP FALSE  \ replace TRUE since no error
      THEN
   THEN
;
: AMENU.TERM ( -- , clean up menus and close window. )
   GR-CURWINDOW @ ?DUP
   IF
      CLEARMENUSTRIP()
   THEN
   MY-MENU EZMENU.FREE
   GR.CLOSECURW
   GR.TERM
;
: AMENU ( -- , do it all)
   AMENU.INIT 0=  \ Everything OK?
   IF
      LOOP.MENU
   THEN
   AMENU.TERM
;
cr ." Enter:    AMENU    to see demo." cr
```

I like to close my program files with a reminder of what to enter to run the program.

EZMenu Glossary

For the following words, the parameter 'ezmenu' refers to the address of an EZMenu structure. All of the addresses passed to and received from these routines are JForth relative addresses unless otherwise specified. All numbering is zero based. The first MenuItem, therefore, has an item# of 0. For the following examples, assume that you have created two menus, as follows:

```
EZMENU MAINMENU
EZMENU EDITMENU
```

EZMENU (<name> --input-- , Create an EZMenu structure.)

EZMENU.ALLOC? (#items ezmenu -- &items | 0)

Allocate memory needed for the MenuItems, the Intuitext items, and the CFA array. The first MenuItem is then linked to the Menu. If any of the allocations fail, a zero is returned. Otherwise the address of the items is returned. You should check the return value before proceeding with any other EZMENU calls.

```
\ Allocate space for 6 menu items.
6 MAINMENU EZMENU.ALLOC?  .
```

EZMENU.CFA[] (item# &ezmenu -- &cfa, Address of cfa)

The EZMenu system maintains an array of CFAs associated with each menu. A CFA is Forths equivalent to 'C's pointer to a function. The array is initially filled with the CFA of NOOP. You can use EZMENU.EXEC to execute the appropriate word after a menupick. The CFA of a word can be obtained by "ticking" a word with the word ' . Any word's CFA can be placed in this array as long as it doesn't take from or leave anything on the stack.

```
\ Set the CFA for menu item 3
: ACT3 ( -- , No parameters allowed.)
    ." Action 3" CR
;
 ' ACT3  3 EZMENU.CFA[]  ( get address )  ! ( store )
```

EZMENU.COMMSEQ! (char item# &ezmenu -- , Set command key.)

You can specify that a menu item be 'picked' by hitting the right Amiga key and a special character together. This provides a handy shortcut to a menu action.

```
\ Set Command Sequence key to 'W'.
ASCII W  3 MAINMENU EZMENU.COMMSEQ!
```

EZMENU.EXCLUDE! (mask item# &ezmenu -- , set auto-exclusion)

Selecting one MenuItem can automatically cause others to become unchecked. See the Intuition Manual, page 6-6, for details. Note that this word also sets the CHECKIT flag.

EZMENU.EXEC (menucode menustrip -- , run menuitem's action)

The MENUCODE is obtained from Intuition. It is placed in the JForth variable EV-LAST-CODE by a call to EV.GETCLASS . The menustrip is the same as the address of the first EZMenu in the list.

EZMENU.FREE (&ezmenu -- , Free memory from EZMENU.ALLOC)

EZMENU.ITEM[] (item# &ezmenu -- &item , item address)

If you want to modify a MenuItem structure for special handling, use this word to obtain it's address.

EZMENU.TEXT[] (item# &ezmenu -- &intuitext , text address)

If you want to modify a MenuItem's associated IntuiText structure, use this word to obtain it's address.

EZMENU.TEXT! (text0 item# &ezmenu -- , Set text for MenuItem)

The text must be a NUL terminated string such as that created by 0" . (That character in front of the quote is a zero. In some fonts this is not obvious.)

```
\ Set menu item text.
0" Write"  3  MAINMENU  EZMENU.TEXT!
```

EZMENU.SET.FLAG (flag item# &ezmenu -- , OR flag with existing)

You can set your own bits in the flags member of a MenuItem structure with this word.

```
\ Put checkmark beside item# 2 .
CHECKED 2 MAINMENU  EZMENU.SET.FLAG
```

EZMENU.SETUP (name0 menu# &ezmenu -- ,Set default values)

This word initializes the values in the Menu, MenuItem, and IntuiText structures associated with an EZMenu. It then links together all the pieces for Intuition to use. It must be executed after EZMENU.ALLOC and before any calls to EZMENU.TEXT! , EZMENU.SET.FLAG, EZMENU.COMMSEQ! , etc. The first parameter is a NUL terminated string that is the name for the menu. The second parameter is the menu's intended position in the MenuStrip. Remember the first menu is number 0. You may want to change the default settings before calling this routine. See below.

EZMENU.SUBMENU! (&submenu item# &ezmenu --)

Set submenu for this item. The example in JD:DEMO_MENUS uses submenus.

EZSUBMENU.SETUP (&ezmenu -- , set defaults and links)

EZMENU.SETITEM (0name cfa char|0 item# &ezmenu --)

Sets the name, cfa and command character for a menu item. Provided for convenience.

EZMenu Default Settings

The EZMenu system uses default settings for many of the Menu and Menuitem parameters. These are kept in variables that you can change before calling the EZMenu routines.

INTUITEXT-DEFLEFT (--- addr ,default left edge of IntuiText item)

MENU-DEFLEFT (--- addr , default left edge of a menu)

MENU-DEFWIDTH (--- addr , default width of a menu)

MENUITEM-DEFLEFT (--- addr , default left edge of a menu item)

MENUITEM-DEFWIDTH (-- addr , default width of a menu item)

Low Level Menu Support

The EZMenu system uses some of these words to perform it's functions. Several of these words will need to be called by your application directly.

The following three words are used by EZMENU.SETUP to initialize the Amiga structures needed to use menus.

```
INTUITEXT.SETUP  ( &intuitext -- , Set defaults for IntuiText)
ITEMNUM()  ( menucode -- item# , parse code from event handler )
MENU.LINKTO  ( menu1 menu2 -- , Make menu2 follow menu1 )
```

A menustrip can be built by linking a number of menus together into a linked list.

```
MENU.MIS># ( subitem# item# menu# -- menunum , Calc MENUNUM)
```

Many Amiga routines use this compounded menu number.

```
MENU.NTH    ( N &menustrip -- &Nth_Menu , Traverse Menustrip )
```

Follows links in menustrip's linked list to find Nth menu.

```
MENU.SETUP   ( name0 menu# &menu -- , Set defaults for Menu)
```

See EZMENU.SETUP for description of parameters.

```
MENUITEM.SETUP ( item# &menuitem -- , Set defaults for MenuItem)
MENUNUM()  ( menucode -- menu# , parse code from event handler )
SUBNUM()  ( menucode -- subitem# , parse code from event handler)
```

The following words make calls to the Intuition library.

```
SetMenuStrip()  ( &window &menustrip -- , Attach menus to window)
```

The menustrip is the address of the first menu in the linked list.

```
ClearMenuStrip() ( &window -- , Remove the menus from a window.)
OnMenu()  ( window menunum -- , Enable part of menu )
```

These routines are used when a menu is currently attached to a window. The menunum can be created using MENU.MIS># .

```
OffMenu() ( window menunum -- , Disable part of menu)
```

Chapter 21
IFF Support

IFF files contain information that can be shared between various programs. Graphics and Animation programs on the Amiga save picture information in IFF files. Thus you can save a picture from a paint program and use it in an animation program. Music programs will save sound samples in IFF files The IFF standard was developed by Electronic Arts.

JForth provides three levels of support for IFF files. The lowest allows you to write a program that can read or write various IFF files. The second level provides an easy system for reading and writing ILBM picture files. The top level provides a rudimentary animation language based on the manipulation and display of images read from IFF files.

IFF pictures can even be combined with free form graphics generated using the graphics library for some very special custom effects. An example application, JSHOW, is included to show you how to open a custom screen and display pictures from a file. If you are not interested in the IFF picture files, you may want to skip ahead to the section on the IFF file format.

We do not provide high level support for 8SVX sample files but this could be written using the IFF tools. We do not imagine that this system will serve all the needs of all the people. We have, however, tried to provide tools that are flexible enough so that others can use them. Our main intent here is to provide you with a starting point for developing your own code. All the IFF source code is in the directory JIFF:. I urge you to print it out, study it, modify it and make it your own.

Description of Files in JIFF:

PICTURES - High level system for loading and displaying IFF pictures. This words could be used as the basis for a bitmapped animation system. You can combine these words with the normal graphics calls for custom effects. If you want to save your resulting animation to give to friends, Clone it!

PIC_EFFECTS - Special effects using Pictures - Wipe and Fadein, Fadeout.

PIC_FLIP - Flip a picture about x,y or diagonally.

ILBM_PARSER - Tools for parsing an ILBM bitmap.

ILBM_MAKER - Tools for writing a bitmap as an ILBM IFF file.

SHOW_IFF - Display tools, application for displaying IFF files.

IFF_Support - This file contains the core words for parsing an IFF file. It has tools for reading and writing chunks, and processing special chunks like 'FORM'.

PACKING - Low level support for packing picture data into ILBM form. Support for run length encoding a Bitmap and converting a CTABLE to a CAMP.

UNPACKING - Routines for unpacking data from an ILBM IFF file.

IFF.J - Definition of BitMapHeader structure and the common chunk IDS.

Tutorial 1 - Displaying Pictures

First let's compile the IFF code and display a simple picture. We provide an IFF picture, 2 brushes and an animbrush in the directory JPics:.

Enter:

```
INCLUDE JIFF:SHOW_IFF
JSHOW JPICS:MOUNTAINS.PIC
```

JSHOW will read your file, open a screen, and display the picture. Click in the top left corner when you are done looking. (The images provided with JForth were chosen for their small size when compressed and not on artistic merit, as you will see.)

Tutorial 2 - The Picture System

Now let's get fancy with bitmaps. We will use the highest level - the "JForth IFF Picture System". As a first step, let's load a picture to use as our background. We can use the same picture of mountains. Enter:

```
INCLUDE JIFF:LOAD_PIC
GR.INIT   \ Open Graphics and Intuition libraries
PICTURE BACKG  ( declare structure )
" jpics:mountains.pic" BACKG $PIC.LOAD? .
```

The second line declared a picture structure that is used to keep track of pictures that are loaded into the system. You can declare as many of these structures as you need. The third line loaded the graphics from the file "jpics:mountains.pic" and stored it "in" the Picture structure called BACKG. A zero should have been returned from this word if everything went OK. The FIRST picture loaded always causes a screen to open with the proper resolution and depth to display that picture. It is important that this first picture be a full screen picture representative of the resolution you will be using in your program.

You should now see your picture. You can move between screens by hitting these key combinations: <left-Amiga-N> to get the workbench screen, <left-Amiga-M> to get other screens. You might now want to shrink your JForth window and pull your workbench screen down a bit (grab it at the top) so you can see both screens. Click in the JForth window, then enter:

```
PICTURE MYSHIP
" jpics:ship.br" MYSHIP $PIC.LOAD? . ( read bitmap )
```

With the above commands, we read the JPICS:SHIP.BR brush into a bitmap and saved the pointer to the bitmap in the structure MYSHIP.

Let's draw our ship in the screen. Enter:

```
20 30 MYSHIP PIC.BLIT
```

Click over to the screen with the mountains on it. You will probably see a rectangular block near the top left of the screen. Inside the block will be the ship. (The word "BLIT" is computerese for Block Transfer of Pixels. This means one image is drawn in another.)

The black rectangle is not a bug. Bitmaps are rectangular and when you just draw them using PIC.BLIT you get the whole thing. Luckily there is a way to copy a bitmap while keeping the

background of the bitmap transparent. This will be more like what you would see when you draw with a brush in a paint program.

To copy a bitmap transparently we need to use a shadow mask of that bitmap. A shadow mask is a special kind of bitmap that has only one real memory plane. It is designed to look as if it has several planes. When you blit another bitmap into it, any color other than 0 will turn on the pixel in the shadow mask. This shadow mask is then used to cut a hole in the destination bitmap. The picture can then be placed into that hole using an OR mode Blit. The picture system will make a shadow mask for you automatically if you try to do a transparent blit.

```
120 40 MYSHIP PIC.TRANS.BLIT
```

If you look at the picture now you should see another ship but without the rectangle.

Drawing a Portion of a Picture

For this next exercise we would like to have two pictures so let's modify the picture called BACKG to look different, then load another copy of our mountains. To change BACKG, let's draw a big rectangle in it. Enter (exactly):

```
4 GR.COLOR!
10 10  310 190 GR.RECT
```

There should now be a big rectangle in the middle of the picture.

Now let's reload up our mountains in another PICTURE for these next exercises.

```
PICTURE MNTNS
" jpics:mountains.pic" MNTNS $PIC.LOAD? .
```

Note: You will **not** see the new mountains picture. We are still displaying the modified BACKG picture.

We can use the picture system to draw only a portion of a bitmap. Suppose we want to take part of the top left of the mountain scene and draw it to the middle of the background. Enter:

```
100 80 MNTNS PIC.PUT.WH
```

This sets the width and height of the region to be drawn from. Enter:

```
120 70 MNTNS PIC.BLIT
```

You should now see the top left corner drawn in the middle of the screen. We can set the corner of our region anywhere in the picture. Enter:

```
200 100 MNTNS PIC.PUT.XY
0 0 MNTNS PIC.BLIT
```

You should now see a 100 by 80 pixel rectangular portion of the lower right of the mountain scene drawn at 0,0 on the screen.

This diagram shows the source region in a picture called Source and the region it is Blitted to in a picture called Destination.

Special Effects - Wipes and Fades

We can do a few special effects that might be useful in constructing a video. Remember the composite output of your Amiga can be plugged into a VCR and recorded! To fade to black and then come back with another picture. Enter:

```
4 BACKG PIC.FADEOUT
PIC-START-BLACK ON   ( make MNTNS start black )
MNTNS PIC.DISPLAY
8 MNTNS PIC.FADEIN
```

This is a gentle way to transition between two scenes. The command PIC.DISPLAY makes the specified picture be visible. [Hacker's note - this works by copying pointers to that picture's bit planes to the open screen's bitmap.] The word PIC.FADEOUT and IN take a time parameter and a picture address. The time parameter is the number of frames to wait before each change in brightness.

In this system we distinguish between the picture being displayed and the picture being drawn into. This allows us to do what is called double buffering which is a way to eliminate some of the jitters in animation. We are displaying the MNTNS picture but we are still drawing into the BACKGROUND picture. Let's draw into the BACKGROUND using the Graphics toolbox then rapidly switch display buffers. Enter:

```
23 45 " Peace On Earth" GR.XYTEXT
```

If you look on the screen, you WON'T see this text. Now enter:

```
BACKG PIC.DISPLAY
```

We can do another effect called a wipe that is used as a transition between pictures. You may want to resize the JForth window then pull down the Workbench screen so that you can see the graphics screen as well. Watch the screen and enter:

```
MNTNS PIC.WHOLE ( reset source window to whole picture )
0 0 2 WIPE_RIGHT MNTNS PIC.WIPE
```

This told the Picture system to "wipe" the MNTNS picture into the current picture being drawn to. The 0,0 was the x,y of the top left corner. The 2 was the number of lines to blit per frame. The WIPE_RIGHT parameter was the direction. You can also choose between WIPE_LEFT, WIPE_UP, and WIPE_DOWN. You can wipe with part of a picture by setting the region with PIC.PUT.WH and PIC.PUT.XY.

Moving a Brush, Restoring the Background

Let's reload the MNTNS picture so that we have a clean slate. Enter:

```
" jpics:mountains.pic" MNTNS $PIC.LOAD? .
MNTNS PIC.DISPLAY
MNTNS PIC.DRAWTO
```

Suppose we wanted to make a brush move smoothly across the screen. We would need to draw it at one location, then draw it again moved slightly in the direction of motion, and so on. If we do this, however, we end up with a trail of brushes. Obviously we must erase one image before we draw the next. How can we do that? There are basically two ways. One is to rebuild the entire image by copying in a fresh image of the background that was saved away, then drawing the brush in the new position. This assumes that you have a copy of the background saved. If that is not that case then you will need method number two.

In this method we save a small portion of the background, just the amount that will be covered when we draw our brush. To erase our brush we can then copy this saved portion back to its original location.

Let's try this with our ship. First we must allocate a bitmap for this backup image. Enter:

```
0 MYSHIP PIC.ALLOC.BACKUP? .  \ must be zero
```

A zero is passed as the first parameter to select between backup image number zero or one. These two backups come in handy when doing double buffering because the brush appears in two backgrounds

Now let's make a backup copy under our brush, then draw the brush. Enter:

```
20 45 0 MYSHIP PIC.BACKUP.NTH  \ no visible change
20 45 MYSHIP PIC.BLIT     \ must be to same X,Y
```

Now to move the brush we must first restore the old background. Enter:

```
0 MYSHIP PIC.RESTORE.NTH
```

Notice that the background is restored.and we can redraw the brush in a new location. Enter:

```
24 53 0 MYSHIP PIC.BACKUP.NTH
24 53 MYSHIP PIC.BLIT
```

By continuing in this manner, a brush can be made to move across the screen.

If you are moving multiple brushes in an image that might overlap, you must follow a simple rule:

For multiple brushes, call PIC.RESTORE.NTH in the opposite order that you call PIC.BACKUP.NTH.

Otherwise you will not restore the image properly.

Cleaning Up

We must now cleanup the allocated memory, close the screen. Enter:

```
MNTNS PIC.FREE
BACKG PIC.FREE
MYSHIP PIC.FREE
```

Whichever picture is currently being displayed will close the screen when freed When finished using the graphics toolbox, we should close the Graphics and Intuition library be entering:

```
GR.TERM
```

For more examples of using the picture system, look at our simple test program: JIFF:TEST_PIC.

Picture System Reference

The Picture System provides easy to use routines for displaying and manipulating images read from IFF ILBM files. It can be loaded by entering:

```
INCLUDE JIFF:LOAD_PIC
```

Error Handling

Many of the routines return an ERROR? flag that is non-zero if there was an error. Your program should always check this flag when it is returned. The most common sources of errors would be if memory could not be allocated for an operation, or if there was a problem with a file. Well written applications should expect these errors to occur periodically and to respond gracefully when they do. Do not simply ABORT because your user might lose valuable work in progress. See the documentation for GOTO.ERROR for tips on handling errors.

Double Buffering

Double buffering is a technique used to achieve smooth animations. The basic sequence of operations is:

```
Display Buffer 0
Draw to Buffer 1
Display Buffer 1
Draw to Buffer 0
Repeat
```

In this manner the viewer always sees a static image while the other image is being drawn.

There are three ways to do double buffering provided by JForth, each with their own advantages and disadvantages. You can also develop your own system if none of these suit your needs.

1) Use the tools in JIFF:DOUBLE_BUFFER. These are demonstrated in JD:DEMO_DBUF. The advantage of these is that they are independent of the PICTURE system, and are very convenient.

2) Use PIC.VIEW to switch displays. This switches very fast because it uses LoadView() and is very convenient when using the PICTURE system. The disadvantage is that it works below the level of Intuition so mouse input may not go where you think it will. Be sure to bring your screen to front before calling this or else your mouse input may go to another screen. To bring the screen that the PICTURE system uses to the front, enter:

```
SIFF-SCREEN @ ScreenToFront()
```

3) Use PIC.DISPLAY to switch displays. This works well with Intuition but is quite slow. It takes about 2-3 frames just to switch displays compared to PIC.VIEW which is virtually instantaneous.

Using your Own Display Screen

The PICTURE system uses the screen pointed to by the variable SIFF-SCREEN. If you load a picture using $PIC.LOAD? and there is no screen, a screen will be opened for you and its address placed in SIFF-SCREEN. If you want to use your own screen, open it before calling $PIC.LOAD? and place your screen address in SIFF-SCREEN.

Clipping with Pictures

There are two types of clipping involved with the Picture system. If you draw to the picture that initiated the opening of the screen, then all graphics operations will be clipped to the BACKDROP window in that screen. This includes calls to PIC.BLIT and other calls like GR.DRAW or GR.TEXT. If, however, you call PIC.DRAWTO to draw to another picture, then only PIC.xxx calls will be clipped. Line drawing using GR.DRAW and similar calls will not be clipped and could result in the trashing of memory. This is because the RastPort of the BACKDROP window has a ClipRect but the RastPort for a picture does not. The PIC.xxx calls are clipped using a custom clipping routine designed for rectangular blits. By setting the variable PIC-CLIPPING OFF you can turn off this custom clipping.

All drawing is directed to the RastPort whose ABSOLUTE address is in the variable GR-CURRPORT. This can be set directly or by using GR.SET.CURWINDOW.

Picture Glossary

JIFF:PICTURE

$PIC.LOAD? ($filename picture -- error? , load IFF picture)

Load an IFF ILBM picture from a file to a picture. The first one loaded will open an appropriately sized screen for display. Make sure, therefore, that the first picture loaded is a full screen picture that is the right resolution for the rest of your animation. You might want to load a title screen first. Set the variable PIC-START-BLACK to TRUE if you want this one to start black. You can then fade in using PIC.FADEIN .

$PIC.SAVE? ($filename picture -- error?)

Save the contents of a picture in an IFF ILBM file for use with other graphics applications. The bitmap will be compressed using run length encoding.

PIC-CLIPPING (-- addr)

When this variable is set TRUE, then PIC.BLIT and PIC.TRANS.BLIT will be clipped to the edges of the destination picture. This may be redundant if drawing to a window with Amiga based clipping.

PIC-START-BLACK (-- addr , variable to control color)

If this variable is set TRUE, then $PIC.LOAD? and PIC.DISPLAY will set the screen to all black. You can then use PIC.FADEIN to make the picture visible.

PIC.?BREAK (-- , OBSOLETE, don't use)

PIC.ALLOC.BACKUP? (backup# picture -- error?)

Allocate a bitmap to use with PIC.BACKUP.NTH. Backup# is 0 or 1.

PIC.ALLOC.SHADOW? (picture -- error?)

Allocate a shadow mask bitmap for use with PIC.TRANS.BLIT. You must also call PIC.CAST.SHADOW before calling PIC.TRANS.BLIT.

PIC.ALLOC.VIEW? (picture -- error?)

Allocate a VIEW for PIC.VIEW.

PIC.BACKUP.NTH (dstx dsty backup# pict --)

Backup part of the image that will be overwritten when we do a PIC.BLIT or PIC.TRANS.BLIT at the same DSTX and DSTY. That image can then be restored using PIC.RESTORE.NTH. This is used mostly when using a moving brush. Backup# is 0 or 1.

PIC.BLIT (xd yd picture -- , blit to x,y)

Blit the bitmap of a picture into the current rastport at xd,yd. The current rastport is the one whose absolute address is in GR-CURRPORT . PIC.DRAWTO can be used to select a picture to BLIT into. By BLITting a series of related pictures you can obtain smooth animation effects.

PIC.BUILD (bitmap picture -- , build a picture from scratch)

If you want to use a bitmap that does not come from an IFF file as a picture, use this word. The bitmap should already be initialized and have an image associated with it.

PIC.CAST.SHADOW (picture --)

Create a shadow mask by ORing each plane of a picture into a single plane. You should call this routine anytime you change a picture that you use with PIC.TRANS.BLIT.

PIC.CLOSEBOX (-- , OBSOLETE, don't use)

PIC.COPY (srcpic dstpic -- , copy bitmaps and color table)

Copy the contents of one picture to another. The second picture must be the same size as the first. Use PIC.DUPLICATE to allocate a same sized picture first if needed.

PIC.DISPLAY (picture -- , display picture by copying bitmaps)

Causes this picture to be the current one displayed. Use this with PIC.DRAWTO for double buffering. Loads pointers to this picture's bitmap planes into the SIFF screen's bitmap.

PIC.DRAWTO (picture -- , make this the destination)

Sets the JForth graphics variable GR-CURRPORT to point to this picture's Rastport (absolute address). Now PIC.BLIT, PIC.WIPE, etc. and all GR.xxx commands will draw into this pictures bitmap. You can draw into one picture while displaying another for a double buffering effect.

Warning: Pictures do not have clipping layers in their RastPorts. Thus any line drawing or other graphics operations may extend beyond the edges of the pictures and overwrite memory. PIC.BLIT and PIC.TRANS.BLIT will continue to be clipped as long as PIC-CLIPPING is set TRUE.

PIC.DUPLICATE? (srcpic dstpic -- error? ,)

Take an empty picture and allocate bitmaps for it the same size as the source picture. Then call PIC.COPY to copy the bitmap contents.

PIC.FREE (picture -- , free all parts of picture)

Free all allocated internal data. This MUST be called when you are completely through using a picture. The picture that is currently being displayed via PIC.DISPLAY will close the SIFF screen when this word is called.

PIC.GET.DEPTH (picture -- depth , number of planes)

PIC.GET.WH (picture -- w h , fetch source w and h)

PIC.GET.XY (picture -- x y , fetch source x and y)

PIC.GET.XYOFF (picture -- x y , fetch dest x and y)

PIC.MAKE? (colrtab|0 #colors deep wide high pict -- error?)

Create bitmaps and other necessary structures for a picture based on input parameters. This is an alternative to $PIC.LOAD?. COLRTAB can be zero or the address of a color table whose contents will be copied to a newly allocated color table.

PIC.OPEN? (picture -- screen | 0)

Open a screen based on the picture.

PIC.PUT.WH (width height picture -- , set source width, height)

Sets the width and height of a rectangular portion of a picture. This is what will be drawn using PIC.BLIT and PIC.WIPE.

PIC.PUT.XY (x y picture -- , set source x and y)

Set the top,left x,y of a rectangle to draw from. Used with PIC.PUT.WH .

PIC.PUT.XYOFF (x y picture -- , set dest x and y)

When this picture is drawn with PIC.BLIT or PIC.TRANS.BLIT, the **destination** x,y will be offset using these values. This can be used to "center" a picture so that the coordinates you pass to PIC.BLIT determine where a certain part of a bitmap will land other than the top,left corner.

PIC.RESTORE.NTH (backup# picture --)

Restore the part of the image saved using PIC.BACKUP.NTH.

PIC.TRANS.BLIT (xd yd picture -- , blit transparently)

Copy a bitmap into the current rastport using a transparent background. This is useful when using brushes from a point program. If you are drawing a head then you just want the round part superimposed over the background. Without transparency, you would get a black rectangle with a head in the middle! This routine uses a shadow mask rastport. The mask is used to punch a hole in the background where there is data in the picture. It then ORs the bitmap with the rastport. The opaque pixels in the bitmap will line up with the black hole in the background. The transparent black pixels in the bitmap will line up with remaining pixels in the background. When you or these you get a nice superimposition of the bitmap over the rastport.

PIC.USE.COLORS (picture -- , apply colors to screen)

Use the color table from a picture in the SIFF screen.

PIC.VIEW (picture --)

Displays a picture by calling LoadView(). This will call PIC.ALLOC.VIEW? just in case it hasn't been called yet. This is faster than PIC.DISPLAY but Intuition will not realize the display has changed. This can cause unexpected results. If, for example, the Workbench screen was showing before this call, mouse clicks will still go to the Workbench even though this picture is showing in front.

PIC.WHOLE (picture -- , reset bounds to use whole picture)

Resets the source x,y and w,h to the full picture boundaries.

JIFF:PIC_EFFECTS

PIC.BRIGHTNESS (level picture -- , scale colormaps)

Set the SIFF screen brightness by scaling the color map in a picture. The range is zero to 16 where 16 is full brightness.

PIC.FADEIN (frames picture -- , fade in from black)

Call PIC.BRIGHTNESS in a loop from 0 to 16. The frames parameter determines how many video frames to wait between successive level changes. A value of 4 is nice.

PIC.FADEOUT (frames picture -- , fade to black)

PIC.NEXT.WIPE (picture -- done?)

Draw next portion of a wipe based on PIC.SETUP.WIPE call. Call this in a loop until it returns true. It doesn't hurt to call it after it's done.

PIC.ROTATE (-- , rotate siff-screen)

Rotate the plane pointers in the SIFF screen. Causes wild color changes. Do it as many times as there are planes if you want to get back to the same color. Use PIC.GET.DEPTH to find the number of planes.

PIC.SETUP.WIPE (xd yd nlines direction picture --)

Set up a picture for a wipe effect. The data will be taken from the source rectangle and drawn to where xd,yd is at the top,left corner. You can specify the number of lines per wipe pass. The more lines the faster the wipe. Try to make it divide evenly into the number of lines total. The direction parameter can be one of four values:

```
WIPE_LEFT  WIPE_RIGHT  WIPE_UP  WIPE_DOWN
```

WIPE_LEFT will cause to wipe to progress from right to left. You may setup several pictures then call PIC.NEXT.WIPE for each one in a loop to have simultaneous wipes happening in parallel.

PIC.WIPE (xd yd nlines direction pict -- , wipe a picture)

Call PIC.SETUP.WIPE then loop on PIC.NEXT.WIPE until done.

JIFF:PIC_FLIP

PIC.FLIP.X (picture --)

Flip a picture horizontally.

PIC.FLIP.Y (picture --)

Flip a picture vertically.

PIC.FLIP.DIAG (picture --)

Flip a picture about a line drawn from two diagonal corners.

IFF File Support

(For a more detailed description of IFF, please see the ROM Kernal Manual Volume 2)

IFF files are made up of "chunks". A chunk has 3 parts:

```
1) Chunk ID - 4 characters, eg. 'FORM', 'BMHD'
2) Chunk Size - in bytes
3) Chunk Data - whatever
```

The chunk ID tells a program what kind of chunk it is. For example, 'CMAP' means that the chunk is a color map for a picture. A chunk ID consists of 4 character packed into a 4 byte integer. The second part, the chunk size, tells you how many bytes are in the data portion. (If there are an odd number of data bytes, a pad byte will be added so that the next chunk starts on an even boundary.) The data portion is where the actual pictures, samples, note lists, etc. are stored.

There are several Chunk IDs which are considered special. These chunks can contain other chunks in their data portion. They are FORM, LIST and CAT. An IFF file consists of one of these 3 chunks containing one or more sub chunks. FORM chunks are the most common of the 3. The data portion of a FORM consists of a FormType followed by a number of subchunks. One common FormType is 'ILBM' which means that the FORM contains chunks that describe an InterLeaved BitMap, or picture. The chunks that describe a picture include 'BMHD', or BitMapHeaDer, which tells you how many pixels high and wide a picture is, how many bit planes it has, where it is positioned on the screen, etc. Another chunk in an ILBM is the 'BODY' that contains the actual pixels of the picture. These are often compressed to save space on the disk.

How JForth Handles IFF files

When you open an IFF file, you really don't know what kind of chunks you will find inside. To read an IFF file, therefore, you must be prepared to handle anything you find.

JForth provides a word called IFF.SCAN that reads the chunk headers and eats its way through an IFF file to see what chunks are there. It uses the chunk size to move from one chunk to the next. Once it has the chunk ID and size it passes these to the deferred word IFF.PROCESS.CHUNK. IFF.PROCESS.CHUNK can then check to see if it is a special type of chunk, ie. a 'FORM', 'LIST' or 'CAT'. This can be done by calling IFF.SPECIAL? which will process the special chunk if it is one. IFF.SPECIAL? returns a flag that tells IFF.PROCESS.CHUNK whether the chunk has already been processed. If not IFF.PROCESS.CHUNK can do whatever it needs to for that chunk. You can set this word to be anything you want and thus control how the IFF file is processed. There are several chunk processors to parse ILBM files, or to print an outline of chunks in the file.

Tutorial 3 - Vectored Parsing of IFF Files

In the previous tutorial, we used the existing ILBM parser to display an IFF picture. Let's now write our own custom parser.

(The parsing of IFF files is done using deferred words. If you are not familiar with DEFER, please see the section on DEFER in this manual.)

IFF.PROCESS.CHUNK is the most important deferred word in this system. Its stack diagram is:

```
IFF.PROCESS.CHUNK ( size chkid -- )
```

Printing Chunk Headers

It is called by IFF.SCAN which is called by IFF.DOFILE. We can write a word to be executed when IFF.PROCESS.CHUNK is called. Let's first write a simple word to print out the header of a chunk. We have a word called .CHKID that will print a packed 4 character chunk ID so let's use it. Enter:

```
: SHOW.CHUNK  ( size chkid -- )
        .CHKID SPACE . CR
;
20 'BMHD' SHOW.CHUNK
```

Now let's use this to examine our MOUNTAINS file. Make sure you are in the same directory as your MOUNTAINS file then enter:

```
' SHOW.CHUNK IS IFF.PROCESS.CHUNK
20 'BMHD' IFF.PROCESS.CHUNK
IFF.DOFILE MOUNTAINS
```

Notice that the chunk in the file is a FORM chunk. Where are the other chunks, the BitMapHeader (BMHD) or the pixels (BODY)? They are nested inside the FORM chunk. To parse an IFF file we need to have a recursive parser. This is easier than it sounds. We have a special word for handling chunks like FORM, LIST and CAT called IFF.SPECIAL? It's stack diagram is:

```
IFF.SPECIAL? ( size chkid -- handled? )
```

If the chunk is a special recursive chunk, this word will handle it and return TRUE. Other wise it will return FALSE. To handle a special chunk IFF.SPECIAL? calls IFF.SCAN which in turn calls IFF.PROCESS.CHUNK . Enough talk, let's show we can use this in our code. Enter:

```
: NESTED.SHOW ( size chkid -- )
        2DUP IFF.SPECIAL?
        IF 2DROP  ( we can ignore it )
        ELSE SHOW.CHUNK
        THEN
;
' NESTED.SHOW IS IFF.PROCESS.CHUNK
IFF.DOFILE MOUNTAINS
```

We should now see all the chunks in the file listed. We have a word already written that does the above using IFF-NESTED to show the recursive nature of the file. ("Now he tells me!") Try:

```
IFF.CHECK MOUNTAINS
```

This word can be cloned if you want it.

Parsing ILBM FORMs

JForth provides tools specifically for parsing an ILBM form. The word $ILBM.PARSE.FILE will scan a file for chunks. The BitMapHeader is copied to a structure called ILBM-Header. This contains information used to decipher the rest of the file. The packed BODY chunk and the CMAP chunk are read into allocated memory, and their pointers left in the variable ILBM-BODY and ILBM-CMAP. GRAB chunks and CAMG chunks are read directly into variables called ILBM-GRABXY and ILBM-CAMG. When the parser is finished you can pull values from these storage locations and build a display. Look in the file JIFF:SHOW_IFF for examples of how this is done.

Also take a look at the file JA:DumpIFF.f which prints the contents of an IFF file for analysis.

IFF Support Glossary

JIFF:ILBM_PARSER

$ILBM.PARSE.FILE? ($filename -- error? , parse an IFF file)

Parse an ILBM file based on whatever is in IFF.PROCESS.CHUNK. Uses $IFF.DOFILE. Use ILBM.PARSE.INIT to initialize IFF.PROCESS.CHUNK if you have changed it and want to use the original ILBM parser.

HEADER>BITMAP (bitmapheader -- bitmap | 0 , allocate bitmap)

Allocate a properly sized bitmap based on the contents of the BitMapHeader structure.

These next few variables and structures are set by the ILBM Parser as it reads an IFF file. Look in here for the information found.

ILBM-BODY (-- var-addr , holds pointer to allocated 'BODY')

ILBM-BSIZE (-- var-addr , holds size of 'BODY')

ILBM-CAMG (-- var-addr , holds viewmodes from any 'CAMG' chunk)

ILBM-GRABXY (-- var-addr , holds packed 16 bit x,y from 'GRAB')

ILBM-CMAP (-- var-addr , holds pointer to allocated 'CMAP')

ILBM-CMSIZE (-- var-addr , holds size in bytes of CMAP)

ILBM-HEADER (-- addr , handy BitMapHeader structure)

The ILBM Parser fills this structure with information from the 'BMHD' chunk. Read the IFF.J file for a list of members.

ILBM.ALLOC.BITMAP (-- bitmap | 0)

Allocate a bitmap of the appropriate size and depth based on the information in the ILBM-Header. This bitmap will be used to receive the bitmap as it is unpacked from the 'BODY' chunk. Calls HEADER>BITMAP. Returns zero if the bitmap could not be allocated.

ILBM.CLEANUP (-- , free any data allocated)

Free BODY and CMAP chunk memory allocated by ILBM.HANDLER.

ILBM.FILL.BITMAP (bitmap -- bitmap | 0)

Unpack the body pointed to by ILBM-BODY into the bitmap. Returns 0 if there is an unpacking error.

ILBM.PARSER (size chkid -- , default handler used to parse ILBM)

IFF.PROCESS.CHUNK is set to call this word by ILBM.INIT. BMHD chunks are copied to ILBM-HEADER , a BitMapHeader structure. BODY and CMAP chunks are read into an allocated memory area whose address is stored in ILBM-BODY or ILBM-CMAP. GRAB chunks are read into the ILBM-GRABXY variable. CAMG chunks are read into the ILBM-CAMG variable. Any other chunks are passed to the deferred word ILBM.OTHER.HANDLER for custom processing.

ILBM.INIT (-- , set vectors)

Set IFF.PROCESS.CHUNK to ILBM.HANDLER and set ILBM.OTHER.HANDLER to IFF.NOT.PROC .

ILBM.MAKE.BITMAP (body bsize bmheader -- bitmap | 0)

Allocate and fill a bitmap based on body and BitMapHeader.

ILBM.MAKE.CTABLE (-- ctable num_colors)

Allocate a color table based on the information in the CMAP chunk. Return two NULLs if there was no CMAP or it couldn't be allocated.

ILBM.OTHER.HANDLER (size chkid -- , handle other chunks)

Deferred word called by ILBM.HANDLER when it sees a chunk it doesn't handle, 'CRNG' for example.

JIFF:ILBM_MAKER

ILBM.HEADER.SETUP (bitmap bmapheader -- , set w,h and depth)

Setup BitMapHeader structure values based on the bitmap.

ILBM.WRITE.BITMAP? (bitmap -- error?)

Write as a BODY chunk to the currently open IFF file.

ILBM.WRITE.ILBM? (bmap ctable ctable# -- error?)

Write a bitmap and a ctable to an IFF file. You must call $IFF.OPEN first then call IFF.CLOSE. This is to be considered as an example program. You will probably want to make a copy of this in another file and modify it to suit your purposes.

$SCREEN>IFF? (screen $filename -- error?)

This handy word pulls the bitmap and colortable from a screen and writes an IFF file.

JIFF:SHOW_IFF

Most of these words use the screen pointed to by the SIFF-SCREEN variable.

$IFF>BITMAP ($filename -- bitmap | 0)

Read a file, allocate a bitmap and load it with the picture in the file. You can use this bitmap as a brush or whatever. You must use FREE.BITMAP from JU:GRAPH_SUPPORT to free this bitmap when done.

$IFF>DISPLAY ($filename -- bitmap | 0 , display iff on screen)

Read an IFF file, open an appropriate screen and display the picture. The screen address will be in the variable SIFF-SCREEN . When done you should call SIFF.CLOSE and then use FREE.BITMAP to deallocate the bitmap you have been given. Look at the source code for JSHOW.

IFF>BITMAP (<filename> -- bitmap | 0 , read IFF file)

Reads filename from input stream and calls $IFF>BITMAP.

IFF>DISPLAY (<filename> -- bitmap | 0 , open screen and display)

Reads filename from input stream and calls $IFF>DISPLAY.

JSHOW (<filename> --)

Read the IFF file and display the picture. Take down the picture when you click the topleft corner or hit a key. This can be cloned for a handy IFF display program.

SIFF-SCREEN (-- addr , variable holding address of screen)

SIFF-WINDOW (-- addr , variable holding address of window)

SIFF.BLACKOUT (-- , black out colors on screen)

SIFF.SHOWIT (-- , Put window display in front for closebox.)

SIFF.USE.CMAP (cmap cmsize -- , use CMAP directly from IFF)

Set colors in SIFF screen based on CMAP.

SIFF.USE.CTABLE (ctable #colors -- , use Amiga CTABLE)

SIFF.CLOSE (-- , Close SIFF screen and window.)

SIFF.WAIT (-- , Wait until CLOSEBOX or Keyboard is hit.)

Low Level Support

JIFF:IFF_SUPPORT

$IFF.DOFILE? ($filename -- error?)

Process file using deferred words. Open the file whose name is on the stack, pull out the chunks and call IFF.PROCESS.CHUNK after verifying that it is an IFF file.

$IFF.OPEN? ($filename -- fileid | 0)

Open a file for IFF.READ and IFF.WRITE. Set IFF-FILEID variable to file-pointer.

.CHKID (chkid -- , print a chunk id as 4 characters)

IFF.BEGIN.FORM? (type -- start-position error?)

Start writing an IFF 'FORM' chunk. An example of type is 'ILBM'. The start position is saved for IFF.END.FORM.

IFF.CHECK (<filename> -- , print chunks)

Set IFF.PROCESS.CHUNK to execute IFF.PRINT.CHUNK then call IFF.DOFILE. This results in a list of the chunks that are in an IFF file. This is a handy tool that could be cloned.

IFF.NOT.PROC (size chkid -- , default for ILBM.OTHER.HANDLER)

Prints a message that a chunk was not processed.

IFF.PRINT.CHUNK (size chkid -- , print chunk id and size)

IFF.PROCESS.CHUNK (size chkid --)

This deferred word is called from IFF.SCAN when a new chunk is encountered in the file. You can set it to your own word for customized parsing of IFF files.

IFF.PROCESS.FORM (size --)

This reads a 'FORM' chunk and processes all of the chunks it finds by calling IFF.PROCESS.CHUNK .

IFF.READ (addr #bytes -- #bytes , read from open IFF file)

This uses the fileid obtained using $IFF.OPEN .

IFF.READ? (addr #bytes -- error?, read from open IFF file)

Calls IFF.READ and returns ERROR? true if the number of bytes read does not the number of bytes requested.

IFF.READ.CHKID (-- size chkid | 0 0)

Read the next 8 bytes in file assuming it is a chunk header. Return 0 0 if an error occurs.

IFF.READ.DATA (dsize -- addr | null , allocate space)

Read DSIZE bytes from the IFF file into an allocated memory area. Return NULL if couldn't allocate. This is handy if you encounter a big chunk.

IFF.READ.TYPE (-- typeid | 0)

Read the next 4 byte from the IFF file. Used by words like IFF.PROCESS.FORM to check the FORM type. Return zero if an error occurs.

IFF.SCAN (-- size , read chunk header and doit)

Read the next chunk header and pass it to IFF.PROCESS.CHUNK then move the file pointer past that chunk's data.

IFF.SEEK (position -- , move file pointer, "seek")

The next read or write will occur at this new position.

IFF.SPECIAL? (size chkid -- done?)

Check to see if the chunk is one of the special type, ie. 'FORM', 'LIST', or 'CAT'. If so process it and return true. This calls IFF.SCAN which can result in recursion. Increments IFF-NESTED to indicate depth of recursion.

IFF.WHERE (-- current_pos , in file)

Where are we currently positioned in file?

IFF.WRITE (addr #bytes -- #bytes , write to open IFF file)

Write data to file opened by $IFF.OPEN .

IFF.WRITE? (addr #bytes -- error?)

Calls IFF.WRITE then checks to make sure the number of bytes written matches the requested number. Note: the stack diagram for this word has changed since JForth V2.0. See the note on incompatibilities at the end of this chapter.

IFF.WRITE.CHKID? (size chkid -- error? , write chunk header)

Write an 8 byte chunk header using IFF.WRITE.

IFF.WRITE.CHUNK? (address size chkid -- error?)

Write complete chunk to current file.

JIFF:UNPACKING

BODY>BITMAP (bodyptr bsize bmap compr -- bmap | NULL)

Unpack a body into a bitmap using given compression mode.

CMAP>CTABLE (cmap ctable #entries -- , unpack)

Convert an IFF CMAP to an Amiga CTABLE array.

UNPACKROW (src dst #src #dst -- src' dst' #src' error?)

Unpack a run length encoded row from source to destination.

JIFF:PACKING

This is a new version of PACKING provided by Martin Kees. It uses a virtual file system to write BODY chunks to a file as they are created.

CTABLE>CMAP (ctable cmap #entries -- , pack)

Convert a CTABLE array to an IFF CMAP.

ILBM.MAKE.BODY (bmap compr -- bodyptr bsize | -1)

Allocate a body memory area then pack the bitmap into it using the desired compression mode. Return -1 if an error occurs. Uses the virtual file system to write to a RAM: based file, then reads it back.

WRITE.BITMAP.BODY { bmap iffile compr -- bodysize | 0 }

Write a bitmap to a file as a BODY chunk. It will be run length encoded if COMPR = 0.

JIFF:PACKING_OLD

This is the old version of the packing code that had problems with highly randomized pictures whose compressed form was larger than the original form. Obsolete.

BITMAP>BODY (bmap bodyptr bsize compr -- bsize'|-1)

Pack a bitmap into a body using given compression mode.

PACKROW (src dst src# dst# -- dst' dst# error?)

Pack a row of data using run length encoding. This could be used for other than picture data!

Incompatibilities with JForth V2.0

There have been some changes that may make code written using JForth 2.0 incompatible with JForth V3.0. In version 2.0, when an error occurred, the deferred word IFF.ERROR was called which

typically caused an abort. This is fine when debugging but is totally unacceptable for a finished application. A properly written application should test for possible errors and handle them gracefully. The previous version did not allow programmers to do that. We felt it was better to correct this problem than to perpetuate a mistake. Unfortunately, some words have been changed. Words that might fail due to insufficient memory or problems with a file now return an error flag. We added a '?' at the end of their names to distinguish them from their original versions and to indicate that they return something of interest. Examples are ILBM.WRITE.BITMAP? and $IFF.DOFILE?. One notable exception to this is IFF.WRITE? which existed in V2.0 but did not return a flag. The stack diagram for this word was actually changed so that it now returns an error flag. I hated doing this but felt it was justified for the purpose of consistency.

Another incompatibility is that when $PIC.LOAD? is first called, the current RastPort is set to that of the backdrop window. The advantage of this is that clipping is active for all graphics. If you then call PIC.DISPLAY for another picture, graphics output will got to that picture as if you had called PIC.DRAWTO for that picture. If you want graphic output to go to a specific picture you must call PIC.DRAWTO explicitly. Be aware that only PIC.xxx calls are clipped after a call to PIC.DRAWTO. See the section in this chapter on clipping for more information.

Chapter 22
Anims and Animbrushes

by Martin Kees

(This facility was contributed by Martin Kees. We at Delta Research are very grateful to Martin for his generosity. Martin also contributed code for the ARexx interface and assisted us greatly with beta testing. Fred Fish disk 516 is all Martin's work. It includes a demo version of a CEL animation program called XL which allows "onion-skin" drawing of Animations. It also includes a fascinating puzzle called Enigma, a loom simulator for weaving design, and more. Watch for more great work from this talented programmer.)

Introduction

Amiga programs use a standard IFF file format for exchanging graphic images. These include still pictures, animations, and animbrushes. This allows you to move images between drawing programs like DeluxePaint, presentation programs like Amiga Vision, image processing programs like Art Department Professional, and other programs.

This toolbox provides routines that allow you to load animations and animbrushes, to play the animations and access the frames of an animbrush. Using JForth will allow you to control and coordinate an animated presentation at a very high level.

You are encouraged to study the anim source code files, modify, and expand the code to meet your own needs. If you wish to do a lot with animation you should consider upgrading your AMIGA in three ways:

1) Fast Memory Expansion

2) Chip Memory expansion to 1 meg with the Fatter Agnus

3) A big fast Hard Drive

All three will make your programming environment much more comfortable as well as decrease your development time.

ANIM Formats

Before we explore the details of the ANIM routines, a little lecture about animation file formats will help you to understand the different ANIM functions. There are many ways to achieve a particular effect. Some are more memory efficient but slower, others are fast but conserve strained memory resources. Some animation effects might best be done with just the JForth PICTURE routines.

There are two formats for storing the graphics information of an animation: an ANIMATION and an ANIMBRUSH. An ANIMATION is the more complicated of the two. Its advantages are that it works well with a double buffered display and it can be decompressed quickly. An ANIMBRUSH has the advantage of ease of changing directions in playback at the cost of slightly slower decompression

times. Understanding the structure of the two formats can help you take advantage of their strong points in your code.

Both formats contain an initial ILBM graphic of the first frame of the animation. The rest of the animation is stored in DELTA chunks which are data chunks that just specify the changes that must be made to a previous frame to generate a new frame. The DELTA chunks for the two formats are interpreted differently for the two types of files.

An ANIMATION DELTA chunk contains the absolute value of new bytes in any changed area of the graphic. The DELTA chunks are setup so that each chunk modifies, not the current, but the previous frame of the animation. When double buffering is used the hidden previous frame is modified by a DELTA chunk to produce the next frame. Two extra DELTA chunks are added to the end of the animation sequence that allow regeneration of the first two frames of the animation so that looping effects can be performed.

An ANIMBRUSH DELTA chunk contains the exclusive OR of the old and new bytes in any changed areas of the display. The DELTA chunks act on the current frame to produce the next frame so that only one complete bitmap needs to be maintained. The last DELTA chunk modifies the last frame to produce the first frame anew for looping. Since the data is an exclusive OR of two consecutive frames, if the same delta is applied to the resulting frame, you get back the original. This makes it easy to generate the frames in forward, backward, or ping pong order.

Compiling the ANIM Toolbox

The ANIM and PICTURE toolboxes have so many routines that they will not fit in the normal JForth dictionary. Luckily it is a simple matter to expand the dictionary space. You can store this expanded Forth wherever you have room for it. Let's assume you want to place it in a directory called TMP:. You will need to substitute the name of your own directory.

Enter in the SHELL:

```
RUN COM:JFORTH
```

Enter in JFORTH:

```
200 #K !    \ allocate a 200K dictionary
SAVE-FORTH TMP:AnimForth
BYE
```

You now have a larger dictionary. Enter in the SHELL:

```
RUN TMP:AnimForth
```

Enter in JFORTH:

```
INCLUDE JANIM:LOAD_ANIM
```

After that compiles, let's save it so we don't have to recompile each time.

```
SAVE-FORTH TMP:AnimForth
```

From now on, you can just run that image and have everything ready for immediate use.

Tutorial 1 - Displaying an ANIM File

(Before proceeding with these tutorials, you should be familiar with the PICTURE IFF system described in chapter 21.)

Before performing any JForth graphics operations, we must initialize the graphics system and open the GRAPHICS and INTUITION library. To do this, enter:

```
GR.INIT
```

This tutorial assumes you have Deluxe Paint III as a source of an animation. If not use any ANIM-5 animation file that's available to you. Enter:

```
CD DPAINT:ANIM
DIR
```

You should see a file called CRY. If the above doesn't work, don't worry. You can use any ANIM for this tutorial, assuming that it fits in your memory.

Animations use a special structure to keep track of all the *cels* in the ANIM as well as other graphical information. This structure is an extension to the PICTURE structure. Let's declare a structure for our ANIM. Enter:

```
ANIMATION CRYANIM
```

Now we can load the animation from disk. This will use the IFF parser to scan the IFF file. The deltas, which describe the differences between successive cels, will be loaded into memory. If the load fails, it will return an error flag on the stack. We print that to make sure the load worked. In an animation application, you should check this flag and respond appropriately. Enter:

```
CRYANIM ANIM.LOAD? CRY   .
```

Since this is the first ANIM loaded, it will open a screen of the proper size and display the first frame. To get back to the WorkBench screen, hold down the <LEFT-AMIGA> key and hit the 'N' key. To get back to the image, hit <LEFT-AMIGA>+M.

Now back in JForth, enter:

```
ANIM_LOOP CRYANIM ANIM.PLAY
```

ANIM.PLAY will start displaying the animation in a loop until you click in the top left corner on the hidden closebox or hit a key on the keyboard.

This Animation played as fast as the ANIM.PLAY routine could generate the next frame. This may be too fast for your particular purpose. The defered word ANIM.DELAY is called within the playback loop after each new frame is displayed. A simple way to slow down the playback would use the Vertical Blank delay:

```
: WAIT20  ( --- )
  20 WAIT.FRAMES
;
' WAIT20 is ANIM.DELAY
```

Note that the delay word must have a stack diagram that neither expects nor leaves anything on the stack.

To remove the animation from memory when finished. Enter:

```
CRYANIM ANIM.FREE
```

Tutorial 2 - ANIM Control and Disk Based ANIMS

Assume we have an animation called SPIN in the current directory. Let's do some explorations with it. First let's create an animation structure for it and load it into memory. Enter:

```
ANIMATION MYSPIN
MYSPIN ANIM.LOAD? SPIN   .
```

You should see the first frame appear. Now jump back to the JForth window using the keyboard combination <LEFT-AMIGA><N>. (Hold down the left amiga key, then press 'N'. You can get back to the Animation with <LEFT-AMIGA><M>.) Click in the JForth window and enter:

```
MYSPIN ANIM.STATS
```

You will see a list of information about the SPIN animation. The value displayed for CURRENT FRAME will be zero based for the first loop of the animation and 1 based on subsequent loops. Enter:

```
MYSPIN ANIM.DISPLAY.NEXT? .
```

You should see the next frame of the animation appear. If you have HISTORY ON you can step through the animation by pressing UP-ARROW RETURN combinations. Note that you will keep looping through the frames of the animation after you have passed the "last" fame. If garbage appears instead then the animation you are viewing does not have a two frame loop ending in the animation file. Flip back to JForth and do ANIM.STATS at different points in the animation. You should note that the sequence of frame numbers change between the first loop and the second. (Note that you can also use ANIM.VIEW.NEXT? which is faster but can have some surprising behavior. See the references to PIC.VIEW for more explanation.)

The ANIMATION structure maintains two pictures. One is displayed and the other is hidden. You can see them by:

```
MYSPIN ..@ an_hiding     PIC.DISPLAY
MYSPIN ..@ an_displaying PIC.DISPLAY ( restores the original)
```

Let's play the SPIN animation from disk. First free the memory version:

```
MYSPIN ANIM.FREE
```

This also closes the screen. Enter:

```
MYSPIN ANIM.DISK.LOAD? SPIN  .  ANIM_ONETIME MYSPIN ANIM.PLAY
MYSPIN ANIM.FREE
```

You will see the animation play through once and stop on the last frame. Depending on the speed of your disk drive and the complexity of the animation results may range from good to poor (from floppy drive). Loading and playing from disk might allow very long animations to be shown or at least previewed.

Tutorial 3 - ANIMBRUSHES

AnimBrushes are like brushes in a paint program, except they have multiple frames or cels, like an Animation. An example might be a bird flapping its wings. For this next tutorial you will need a still picture and an AnimBrush. You could use the ones that came with JForth or you could make your own. Let's assume you now have a background picture that we will call BACKG and an AnimBrush brush that we will call BIRD. (It doesn't have to be a bird but we have to call it something!) It is best if they are made with the same pallette, otherwise the AnimBrush colors may look odd.

Although AnimBrushes could be loaded into their own screen, they are more useful when drawn on top of another image. Let's, therefore, load the background picture first so that it defines the screen. The filenames used are for the images on the JTools disk. You may substitute your own files. Enter:

```
PICTURE BACKG      \ declare a picture structure
" jpics:mountains.pic" BACKG $PIC.LOAD? . \ load image
```

Now let's define the AnimBrush stucture and load an AnimBrush file:

```
ANIMBRUSH BIRD
" jpics:bird.anbr" BIRD $ABR.LOAD?.
```

Since a picture was being displayed at the time of the load you saw no change in the display. If nothing was being displayed then a screen would open and display the first frame of the animbrush. You may want to drag the WorkBench screen down a bit so that you can see the graphics and the JForth window at the same time.

An AnimBrush can be blitted onto the screen just like brushes, enter:

```
10 20 BIRD ABR.BLIT
```

The special thing about AnimBrushes is that you can advance to the next frame and blit that. Enter:

```
BIRD ABR.ADVANCE
10 20 BIRD ABR.BLIT
```

If you don't want the full rectangle of the brush, you can blit transparently. Let's move to new x,y coordinates so we can see the difference. Enter:

```
BIRD ABR.ADVANCE
95 14 BIRD ABR.TRANS.BLIT    \ note .TRANS.
```

Use the <UP-ARROW> key to reenter the two previous commands several times. Notice that the new blits overlap the previous blits. Techniques to prevent this by saving the background and restoring are described in the PICTURE tutorial. The same techniques can be used with AnimBrushes. Actually, almost any PIC. call can be used with AnimBrushes including wipes and other effects. Don't however, use PIC.FREE or PIC.LOAD with AnimBrushes!

Now let's define some words that will help us further explore AnimBrushes. Enter:

```
: SHOW.BIRD ( -- , display current frame of bird )
   10 20 BIRD ABR.BLIT
;
: NEXT.BIRD  ( -- , show next cel of Bird )
   BIRD ABR.ADVANCE
   SHOW.BIRD
   BIRD ABR.GET.FRAME . CR? \ print where we are now
;
```

ABR.GET.FRAME returns the frame, or cel, of the AnimBrush currently ready to blit. We can force an AnimBrush to a particular frame using ABR.GOTO.FRAME. Enter:

```
0 BIRD ABR.GOTO.FRAME  \ move to first frame
SHOW.BIRD   \ show it
NEXT.BIRD   \ show next frame
3 BIRD ABR.GOTO.FRAME
SHOW.BIRD
```

If you tell ABR.GOTO.FRAME to a frame beyond the end of your brush, it will just print a message to that effect and do nothing. You can tell how many cels you have by reading it out of the AnimBrush structure. Enter:

```
BIRD S@ ABR_CELS .  \ print total number of cels
```

Let's now define a word that will continuously advance the AnimBrush. Enter:

```
: PLAY.BIRD ( -- )
```

```
      BEGIN
         NEXT.BIRD
         6 WAIT.FRAMES \ wait 6 video frames, 6/60 seconds
         ?TERMINAL
      UNTIL
   ;
   PLAY.BIRD
```

Notice that the bird is advancing and that the numbers go up to the highest frame then start over again at zero. Let's reverse the direction of play. Hit <RETURN> to stop PLAY.BIRD and enter:

```
BIRD ABR.REVERSE
PLAY.BIRD
```

If you would like the brush to PINGPONG, or go both directions, enter:

```
ABR_PINGPONG BIRD S! ABR_FLAGS
PLAY.BIRD
```

Notice the numbers go up to the maximum then back down to zero, etc. To get back to the normal mode, enter:

```
ABR_LOOP BIRD S! ABR_FLAGS
PLAY.BIRD
```

When you're done using these images don't forget to enter:

```
BIRD ABR.FREE
BACKG PIC.FREE
```

To further explore this toolbox, look in the directory, JANIM:TESTS. It contains test programs that can also serve as simple examples. By studying these files, you will see how to create animbrushes from two pictures using ABR.BUILD?. You can then append other pictures to the end using ABR.APPEND.CEL?. You can also edit the internal structure of animbrushes using ABR.DUP.CEL? , ABR.DELETE.CEL? and ABR.REPLACE.CEL?.

Animation Tips

If you need to edit an animation, you can convert it to an animbrush using ANIM>ANIMBRUSH?, edit it, then convert it back using ANIMBRUSH>ANIM?.

In an application, you can bring the animation to the front using ScreenToFront(). The screen address is stored in the variable SIFF-SCREEN so enter:

```
SIFF-SCREEN @ ScreenToFront()
```

ANIM Support Glossary

ANIM IFF tools

These words are used in reading ANIM-5 IFF files.

$ANIM.LOAD? ($filename animation --- error?)

Load animation for playback. The main load routine. IF an_flags set to ANIM_DISKMODE then a disk mode load will occur.

$ANIM.DISK.LOAD? ($filename animation --- error?)

The ANIM_DISKMODE flag is set and $ANIM.LOAD is called. The animation will be kept on disk and read as it is played. This is handy for very large ANIMs.

$ANIM.PREP? ($filename --- error?)

Setup of variables and animation filename previous to an $ANIM.SCAN? .

$ANIM.SAVE? ($filename animation -- error?)

Saves the animation to the file. You cannot save a DISKMODE animation with this word.

$ANIM.SCAN? ($filename --- error?)

Scans the anim file for delta and anim-header chunks.

ANIM.BLIT (x y animation --)

Animations contain two pictures that are needed to reconstruct the images using double buffers. This word will blit the currently picture to the destination RastPort.

ANIM.DISK.HANDLER (size chkid --)

Reads DLTA chunks and collects the FSEEK and size data in lists to be able to access the chunk from disk.

ANIM.DISK.LOAD? (animation <filename> -- error?)

For disk based loading with filename in the input stream. This cannot be used in a colon definition. Use $ANIM.DISK.LOAD? instead.

ANIM.HANDLER (size chkid -- , handles ANIM specific chunks)

Reads DLTA chunks and allocates memory for them.

ANIM.LOAD? (animation <filename> -- error?)

For memory mode loading with filename in the input stream. This cannot be used in a colon definition. Use $ANIM.LOAD? instead.

ANIM.PARSER (size chkid -- , recursively parse ANIM)

Used by $ANIM.SCAN to collect info about ANHD and DLTA chunks present in the file. Precollecting this data allows more efficient memory allocation for the ANIM file.

ANIM.READ.ANHD? (size --- error?)

Reads an Anim-Header chunk into ANIM-HEADER from current IFF file. Checks for correct size.

ANIM.SAVE? (animation <filename> -- error?)

This cannot be used in a colon definition. Use $ANIM.SAVE? instead.

Saves the animation to the filename from the input stream.

ANIMATION Words

These are used to display ANIMATIONs.

ANIM.ADVANCE? (animation -- error? , advance to next frame)

Calls ANIM.APPLYDELTA or ANIM.APPLYDISKDELTA as needed to generate the next frame. The HIDDEN and DISPLAYING buffer pointers are then switched. Will loop around at end.

ANIM.APPLYDELTA (anim --)

Using the hidden pic buffer applies the current memory based delta. Then bumps the delta pointer looping it to 1 if at the last delta. Does not swap the hidden and displayed buffer pointers or cause the new data to be displayed.

ANIM.APPLYDISKDELTA? (anim -- error?)

Same as ANIM.APPLYDELTA but obtains the data from disk.

WARNING: to use this word you need to previously open the disk based file with ANIM.DISK.OPEN. At the present time only ONE disk based anim can be open. Since the IFF routines are used to read the data you also must not do an IFF read or write operation while the file is open.

ANIM.CHECK (animation -- , abort if bad)

Looks for the correct value in the an_key field.

ANIM.DELAY (---)

A deferred word called by ANIM.PLAY between frames. It can be used to slow down the playback or add syncronization effects.

ANIM.DISK.OPEN? (animation -- error?)

Opens the original file read by the ANIM.DISK.LOAD routine for disk based animations. If you gave a directory relative filename at the original load, you must be in the same current directory to reopen it.

ANIM.DISK.CLOSE (---)

Closes the file.

ANIM.DISPLAY.NEXT? (animation -- error?)

Calls ANIM.ADVANCE? to generate the next frame then displays it using PIC.DISPLAY. This is slower then ANIM.VIEW.NEXT? but preserves the Intuition Screen order.

ANIM.FREE (animation -- , free all parts of animation)

Releases all the allocated memory for the animation and clears the animation struct.

ANIM.GET.DEPTH (animation -- depth)

Returns the number of bit planes in the animation.

ANIM.LAST.FRAME? (animation --- flag)

Returns a TRUE flag if at the last frame of an animation. This is valid ONLY IF the animation contains the typical two extra DELTA chunks for looping. Otherwise returns TRUE on the 3rd from last frame. Problem if the anim has less than 3 frames.

ANIM.PLAY (loopflag animation ---)

Loopflags are ANIM_LOOP (TRUE) which plays the animation in a loop, or ANIM_ONETIME (FALSE) which plays up to the "LAST" frame.

ANIM.STATS (animation ---)

Displays info about the current state of the animation.

ANIM.VIEW.NEXT? (animation -- error?)

Calls ANIM.ADVANCE? to generate the next frame then displays it using PIC.VIEW. This is much faster then ANIM.DISPLAY.NEXT? but can cause some confusion because it overlays the Intuition Screen display using low level code. See PIC.VIEW.

ANIMATION (<animname> ---)

Creates a structure in the dictionary for an animation.

ANIMBRUSH Words

Load and display AnimBrushes.

$ABR.LOAD? ($filename animbrush --- error?)

Since the first IFF load routine used opens a screen, you should load your backgroud picture first so that the display mode can be set. After loading the default direction is set to ABR_FORWARD and the mode flag is set to ABR_LOOP. See ABR.REVERSE.

$ABR.SAVE? ($filename animation -- error?)

Saves the animbrush to the file.

ABR.ADVANCE (animbr --)

Applies current DELTA to the animbrush then updates the delta pointer depending on the current state of the direction and mode flags.

ABR.BLIT (x y animbr ---)

Blits the animbrush into the current rastport then calls ABR.ADVANCE to generate the next frame. A simple PIC.BLIT will do as well but won't advance the frame. The PIC.BLIT call will work since the start of the animbrush structure is a PICTURE struct.

ABR.CHECK (animbr -- , abort if bad , used internally)

ABR.FREE (animbr -- , free all parts of animbrush)

ABR.GET.FRAME (animbr --- current-frame)

Returns the current frame number (0 1 2 .. N) of the bitmap state of the animbrush. Uses the current direction to deduce the current frame.

ABR.GOTO.FRAME (frame animbr ---)

Cycles in the current direction until the asked for frame is generated.

ABR.LAST.FRAME? (animation --- flag)

Returns a TRUE flag if at the last frame of an animbrush.

ABR.LOAD? (animbrush <filename> -- error?)

Reads filename from input then calls $ABR.LOAD. This cannot be used in a colon definition. Use $ABR.LOAD instead.

ABR.REVERSE (animbr ---)

Reverses the direction of the animbrush. That is the order in which the frames are generated by ABR.ADVANCE is reversed. ABR.REVERSE does NOT change the current bitmap. Since we allow forward and backward modes we must deduce the correct DELTA to use if we want to change directions.

ABR.SAVE? (animation <filename> -- error?)

Saves the animbrush to the file name from the input stream.

ABR.STATS (animbrush ---)

Displays info about the current state of the animbrush.

ABR.TRANS.BLIT (x y animbr ---)

The PIC.TRANS.BLIT word only casts the bitmap into its shadow on the first call. Since our bitmap is changing from frame to frame ABR.TRANS.BLIT recasts the shadow at each frame.

ANIMBRUSH (<animbrname> ---)

Creates a structure in the dictionary for an animbrush.

CONVERSION Words

Support for various conversion and animation editing code for EOR delta encoding used in Animbrushes. A true error? flag can be returned if there is insufficient memory for the operation. For examples of their use, see “JAnim:tests/test_conversion.f”.

ABR.APPEND.CEL? (picture animbr -- error?)

Appends a picture to the end of an AnimBrush. Must be same size as the animbrush.

ABR.BUILD? (pic0 pic1 animbr -- error?)

Builds a two frame animbrush from the two pictures. The width and height of the pictures should be set to the size of the entire bitmap of the pictures. You can use PIC.WHOLE to ensure this. The two pictures must be equal in size. The color map of pic0 is used in the generated animbrush.

ABR.CUR.DELTA (abr -- delta-address)

Returns the address of the current (next) delta

ABR.DELETE.CEL? (cel# animbr -- error?)

Deletes given cel from animbrush. An error can occur if memory cannot be allocated for the new deltas.

ABR.DUP.CEL? (cel# animbr -- error? , inserts a duplicate cel)

Adds a new cel to animbrush that is a duplicate of the given cel.

ABR.MAKE.DELTA (pic0 pic1 -- delta | 0)

Using the two given pictures makes an EOR type delta which is used in AnimBrushes. If the Delta cannot be allocated, this will return zero. You must free the delta when you are done using FREEBLOCK unless you add it to an AnimBrush. In that case it will get freed when you free the AnimBrush The pictures must be the same size.

ABR.NORM (animbr --)

Changes direction to FORWARD and mode to LOOPING, saving previous state.

ABR.REPLACE.CEL? (picture cel# animbr -- error?)

Modifies the cel to be the given picture. Picture must be the same size as the animbrush. Calculates the needed changes in the deltalist.

ANIM.MAKE.DELTA (pic0 pic1 -- delta | 0)

Using the two given pictures makes an ABS type delta which is used in Animations. If the Delta cannot be allocated, this will return zero. You must free the delta when you are done using FREEBLOCK unless you add it to an Animation. In that case it will get freed when you free the Animation. The pictures must be the same size.

ANIM>ANIMBRUSH.PARTIAL? (first_cel# last_cel# anim animbr -- error?)

Convert part of an Animation to an AnimBrush.

ANIM>ANIMBRUSH? (anim animbr -- error?)

Constructs an AnimBrush from all of an initalized Animation. This calls ANIM>ANIMBRUSH.PARTIAL? with a first cel# of 1, and a last cel# of 1. This will go through all cels and wrap around to the beginning.

ANIMBRUSH>ANIM.PARTIAL? (first_cel# last_cel# animbr anim -- error?)

Convert part of an AnimBrush to an Animation.

ANIMBRUSH>ANIM? (anim animbr -- error?)

Constructs an Anim from all of an initalized AnimBrush. This calls ANIMBRUSH>ANIM.PARTIAL? with a first cel# of 1, and a last cel# of 0. This will go through all cels and wrap around to the beginning.

ENCODECOL (blk ht --)

Encodes a column in VSkip format in Delta-work @.

EORplane (ABSpl0 ABSpl1 ABSdeltabuff w h --)

Calculates and rotates an EOR type deltaplane.

MAKE.ABSDELTA (pic0 pic1 --)

Creates an absolute style delta for use in ANIMs. Delta stored at DELTA-WORK. Used in ANIMBRUSH>ANIM.

LOW LEVEL support words

These words are used internally by the Animation system. You would not normally call these directly. They are documented here in case you need to make internal modifications or to extend the system.

ADD.YTABLE.USER (ytable --)

Increment the YTable user counter. See FREE.YTABLE

ALLOC.YTABLE (byteoffset linesize -- ytable | 0)

Creates or reuses a YTable. Fails if no memory available or no room is left in list. The constant MAX_Ytabs determines the memory reserved for the list.

ALLOC.YTABLE.TRACKER (-- tracker | 0)

Called automatically by the first request for a YTABLE. Won't hurt anything if done again. A YTABLE is a multiplication table used by the ANIM and ABR routines to optimize the decoding of deltas.

CountSames (buff length -- SameCount)

Low-level word to scan a delta plane for same runs.

CREATE.YTABLE (byteoffset linesize --- ytable | 0)

Allocates and calculates a multiplication table for the decode routines.

DECODE_VKPLANE (inDELTA outBITPlane ytable ---)

ASM code to decode a bitplane of info from a DELTA chunk. This routine expects the DELTA to consist of the absolute byte values in vertical byte skip form. Used by ANIMATION files.

DECODE_XORVKPLANE (inDELTA outBITPlane ytable ---)

ASM code to decode a bitplane of info from a DELTA chunk. This routine expects the DELTA to consist of EOR byte values in vertical byte skip form. Used by ANIMBRUSH files.

FINDOP (buffptr len --- cnt op)

Main low-level word for VSkip encoding of a rotated EOR encoded buffer.

PUSHBYTE (byte memblk --)

Pushes byte to allocated memoryblock

FIND.YTABLE (byteoffset linesize --- ytable | 0)

Checks if an appropriate YTable is available for shared use. If so returns its address, if not returns 0.

FREE.YTABLE (ytable ---)

Frees memory for a YTable if not marked by another user. Last free deallocates the ytable list as well.

FREELIST? (listaddr --)

Listaddr is the relative address of an allocated memory block obtained by a ALLOCBLOCK call. This block contains addresses of a series of memblock addresss. FREELIST? parses the list and does a FREEBLOCK on each memblock in the list, then does a FREEBLOCK for the list.

5 **SCANFORZU (dataaddr length -- Z U)**

Low-level word to scan a delta plane for zero and unique runs.

Chapter 23
ARexx Support

The JForth ARexx support was written by Mike Haas, Martin Kees and Phil Burk. We at Delta Research would like to thank Martin for his generous contributions.

Note: You must have ARexx installed in your system for these tools to be useful. ARexx is a native component of AmigaDOS 2.0 or later. If you are using AmigaDOS 1.3 or earlier, you must purchase ARexx and install it.

What is ARexx?

ARexx is a programming language that can communicate with other applications. Using ARexx, for example, one could request data from a database application and send it to a spread sheet application. To support this facility, an application must be "ARexx compatible" by being able to receive commands from ARexx and execute them. A database program might have commands to search for, retrieve, and save data. MicroFiche Filer is an example of a database program with an extensive ARexx command set. A text editor might have ARexx commands corresponding to its editing command set. The Textra editor supplied with JForth has an extensive ARexx command set, which can be used with JForth to provide an 'integrated programming environment' (see the 'Integrating Textra and JForth' section in this chapter for details). AmigaVision, the multimedia presentation program, can control other programs using ARexx. ARexx can increase the power of a computer by combining the capabilities of various programs. Because of the wide acceptance of ARexx, Commodore includes it with Version 2.0 of AmigaDOS.

If you are developing an application, it is highly recommended that you provide ARexx compatibility, if appropriate. The JForth ARexx Toolbox provides the basic tools needed to call ARexx. It also provides some higher level tools for implementing a command parser, and for manipulating ARexx variables.

ARexx was developed for the Amiga by William Hawes. It is based on the REXX language described by M.F. Cowlishaw in *The REXX Language: A Practical Approach to Programming* (Prentice-Hall 1985)

Description of Files

The JForth support for ARexx is in a directory with the logical name "JRX:".

These files comprise the primary Arexx support files:

ARexxCalls.f = JForth calls to ARexx library

ARexxTools.f = tools to simplify the use of ARexx

MakeArexxMod = Make ARexx module, used when JForth is built

RXUnderKey.f = Install ARexx listener under KEY (eg. for Textra)

RexxClude.f = Support for Textra to ARexx interface

RexxVars.f = Calls for manipulating Rexx variables RexxView.f = An application that monitors ARexx messages for debugging

There are a number of test programs supplied as examples of using ARexx from JForth. When the ".f" test files are compiled, they will print instructions to the screen. The ".rexx" and ".srh" files are ARexx scripts. The test files are in JRX:TESTS and are:

sample_rexx_host.f = sample Rexx host application. Use this as a model for your own applications.

drive.srh = rexx script to drive above sample

macro1.srh = rexx script callable as a macro from above sample

Test_RXTools.f = simple example of controlling JForth from ARexx

ToJF.rexx = sends commands to Test_RXTools.f

vartestf.f = JForth program for testing RexxVars.f

testrvi.rexx = Rexx Script for testing RexxVars.f

Low Level ARexx Support

ARexx structures and constants are defined in "include" files. These have been precompiled into a module called AREXXMOD. To access this module from your program, enter:

```
GETMODULE AREXXMOD
```

JForth compatible calls to the ARexx library are also provided. These routines are documented in the ARexx manual. ARexx is somewhat unusual in that some of the routines return multiple values. Special tools had to be added to the JForth CALL facility to support these. To load these words, enter:

```
INCLUDE JRX:AREXXCALLS.F
```

To see the stack diagram of words in the library, use FILE?. For example:

```
FILE?  STCTOKEN()
```

will show you the following definition:

```
: StcToken()  ( string -- token scan length quote )
   call>abs RexxSysLib_lib StcToken  TuckA1 TuckA0 TuckD1
;
```

ARexx Toolbox Tutorial

[Please familiarize yourself with the ARexx manual before beginning this tutorial. ARexx must be installed for this to work.]

To simplify the development of ARexx compatible programs, we have provided an optional toolbox. This toolbox has routines for initializing the ARexx system, interpreting commands and argument strings, sending messages to ARexx, and cleaning up afterwards. To load this toolbox, enter:

```
INCLUDE JRX:AREXXTOOLS.F
```

To communicate with ARexx, we need to open the RexxSysLib library, create a message port, and initialize a message. When we open a message port, we have to give it a name that will serve as an "address" for ARexx messages. If we have an application called "JAZZ", we can do all of the above with a single call. Enter:

```
: JAZZ.INIT.RX  ( -- error? , setup ARexx )
   0" JAZZ" RX.INIT
;
```

Note that we used 0" instead of just " because ARexx and AmigaDOS use NUL terminated strings. A NUL terminated string is a string of ASCII characters ending with a zero. This is different than a Forth string that begins with a count byte equal to N, followed by N characters. When looking at stack diagrams, parameters like 0NAME and 0STRING are NUL terminated strings. Parameters like $NAME and $STRING are Forth style strings. See 0COUNT >DOS DOS0 and 0" in the Forth glossary for more information.

The stack diagrams for the RX.xxx words can be found in the glossary in this chapter.

We put the call to RX.INIT in a colon definition for two reasons. One is so that the 0NAME would be permanently compiled in the dictionary instead of being on the PAD. Strings defined outside of a colon definition go on the PAD and are soon overwritten. The second reason is because when you write applications, all of the code should be in colon definitions and not just called when the file compiles. For the purposes of the rest of this tutorial, however, we will often just type things in directly because it is easier when experimenting. Now enter:

```
JAZZ.INIT.RX  .
```

The number printed should be zero. It could be equal to the constants ERR_INSUFFICIENT_MEMORY or RXERR_NO_LIBRARY if either of those errors occurred. Checking for RXERR_NO_LIBRARY is one way to determine whether ARexx is installed on a system.

Now let's try sending a message through ARexx to our program. Make sure that you have a Amiga DOS Shell or CLI window open before executing the next command! Enter:

```
RX.WAIT.MSG
```

Notice that JForth has not returned from this word. It is waiting for a message from ARexx. Now click in the Shell window and enter an ARexx command. Pay close attention to the single and double quotes. Enter in the **Shell** (NOT in JForth):

```
RX ' address "JAZZ" "play song" '
```

Notice that JForth has now returned with 1 item on the stack which is the message from ARexx. Notice also that RX in the Amiga Shell window has not returned. RX is waiting for its message back so don't lose the message in JForth.

What does the RX command mean? When you enter an RX command in the shell, you can give as a parameter either an ARexx script filename or an ARexx command delimited by quotes. We used the ADDRESS command to send the message "play song" to the address "JAZZ". Let's now look in JForth to see if we got it. Enter:

```
0 OVER RX.ARG[] \ get address of 0th argument
@ IF>REL        \ fetch it, convert to relative address
0COUNT TYPE     \ print it
```

ARexx messages have up to 16 arguments that can be referenced using RX.ARG[]. They generally contain the absolute address of an ARexxString so we must convert it to relative before using it. The message printed should have been "play song" from our RX command.

Now we must reply to the message so that RX can continue. Enter:

```
DUP .  \ see that our message is still there
RX.REPLY.MSG
```

Let's look at another way of getting messages. Enter:

```
RX.GET.MSG DUP . \ should be zero
0SP
```

Zero means that there were no messages for our program. Let's send another message from the Shell. Enter in the Shell (NOT in JForth):

```
RX ' address "JAZZ" "12345 howdy" '
```

Let's see if we can pick up that number in JForth. Enter in JForth:

```
RX.GET.MSG DUP .          \ should be non-zero
0 OVER RX.ARG[] @ IF>REL \ get 0string
RX.PARSE.NUM .S           \ extract number
```

You should now see four numbers on the stack. The first is our message address, followed by 12,345 (!), followed by the address of the remainder of the string, and a -1 = TRUE. Let's take three of them off of the stack.

```
. \ -1
0COUNT TYPE \ howdy
.  \ 12,345
```

Now before we reply to the message, let's pretend we detected an error that we want to tell ARexx about. The error could be due, for example, to a number out of range, an illegal command, or a song name that JAZZ doesn't know how to play. The numbers that you use are decided upon by you. Let's use 23 for now. To pass an error number back to ARexx, enter:

```
23  RX-RESULT1 !
RX.REPLY.MSG
```

RX should print that the command returned '23'.

An application that supports ARexx commands is called an ARexx host. Most ARexx hosts will need to implement a command set that can be used by other programs. Each command will have a name, a particular set of arguments, and an associated Forth function. We have simplified this process for you. Suppose that we have a very small command set: one word that prints dollar signs, and another that prints a string and a number. These are trivial routines. Your application can have many powerful routines but these two will serve as examples.

First let's define our command functions.

```
: DOLLARS  ( N -- )
   0 DO ASCII $ EMIT LOOP CR
;
: TYPE.  ( ADDR COUNT N -- )
   . CR
   TYPE CR
;
```

Test them by entering:

```
5 DOLLARS
" World Peace" count 123 TYPE.
```

Now lets define a word that, sets up the command interpreter.

```
: JAZZ.SETUP  ( -- )
   2 RX.ALLOC.CTABLE  \ just two commands
   IF
      " DOLLARS" " N" 'C DOLLARS RX.ADD.COMMAND
      " TYPEDOT" " SN" 'C TYPE. RX.ADD.COMMAND
   ELSE
      ." Couldn't allocate table!" cr
   THEN
;

JAZZ.SETUP
```

The stack diagram for RX.ADD.COMMAND is:

```
RX.ADD.COMMAND ( $name $format cfa -- )
```

The format is a string of 'N's and 'S's. For each N, an number will be parsed and passed to your function. For each S, an ADDR and COUNT of a string will be parsed and passed. The stack diagram of your functions must correspond to the format that you give.

There are several ways to use the command table. See RX.EXEC.MSG below. For a simple test we can use RX.SLAVE.SAFE. This will terminate when we touch the keyboard which is handy for testing. Enter:

```
RX.SLAVE.SAFE
```

Now enter in the Shell (NOT in JForth):

```
RX  ' address "JAZZ" "DOLLARS 10" '
RX  ' address "JAZZ" "TYPEDOT hello 123" '
RX  ' address "JAZZ" "DOLLARS 25" '
```

Notice the messages in the JForth window. Now hit <RETURN> in JForth to stop RX.SLAVE.SAFE. When we are done with our command table, we should free it by entering:

```
RX.FREE.CTABLE
```

Now let's finish by entering:

```
RX.TERM
```

Instead of entering ARexx commands directly from the Shell, we could use scripts. Study the test programs in JRX:TESTS for more examples. Pay special attention to JRX:TESTS/SAMPLE_REXX_HOST.F.

ARexx Toolbox Glossary

These words are compiled from JRX:ARexxTools.f

RX-SAVE-MSG (-- addr , saved from RX.GET.MSG)

If your program aborts before getting a chance to call RX.REPLY.MSG, you can get the last message from this variable and reply to it. The use RX.FLUSH.MSG repeatedly until the Rexx program finishes.

RX.ADD.COMMAND ($name $format cfa --)

Add a command to the command table. You must call RX.ALLOC.CTABLE first. The format is a string of 'N's and 'S's. For each N, an number will be parsed and passed to your function. For each S, an ADDR and COUNT of a string will be parsed and passed. The stack diagram of your functions must correspond to the format that you give.

: RX.ALLOC.CTABLE (n -- table | 0)

Allocate table for commands. Call RX.FREE.CTABLE when finished using table. Returns zero if not enough memory.

RX.ARG[] (n rexxmsg -- arg-addr , index into argument array)

Calculate address of an argument in a rexx message. If the argument is a string it will be stored as an absolute address so use IF>REL after you fetch it and IF>ABS before storing a string.

RX.EXEC.LINE (0arg -- error?)

Looks up command in table, parses arguments based on format, and passes as parameters to a Forth function specified by RX.ADD.COMMAND. Error? will be zero, RXERR_ILLEGAL_COMMAND, RXERR_ILLEGAL_PARAMETER or the contents of RX-ERROR as set by your code.

If the word whose CFA you pass in RX.ADD.COMMAND has the wrong stack diagram and leaves something on the stack or eats too much from the stack, you will get an error message and an abort. Please then fix the word for that command. This is considered a serious programming error thus we abort instead of returning a flag.

RX.EXEC.MSG (rexxmsg --)

Pass argument 0 to RX.EXEC.LINE then sets RX-RESULT1 to ERROR?.

RX.FIND.COMMAND (addr cnt -- index true | false)

Look up command in table.

RX.FLUSH.MESSAGES (-- #msgs)

Reply to all messages not yet processed. Sets RX-RESULT1 = 10.

RX.FREE.CTABLE (-- , free allocated table)

Free table allocated by RX.ALLOC.CTABLE.

RX.GET.MSG (-- rexxmsg | 0 , if any are ready)

Be sure to reply using RX.REPLY.MSG when done with message.

RX.INIT (0hostname -- error?)

Open RXSysLib library, create message port, and create a RexxMsg for use if needed.

RX.JFORTH.INIT (-- error? , initialize port as "JFORTH")

RX.KILL.SCRIPT (rexxmsg -- , send CTRL-C to rexx script)

RX.PARSE.NUM (0string -- num 0scan true | false)

RX.PARSE.STRING ($format 0string -- ...params... 0left true | false)

Params will contain N for each 'N' in $format, and ADDR COUNT for each 'S'. If an error is encountered, a false is returned.

RX.PUT.ACTION (action -- , set in rx-message-ptr)

Set action field in RX-MSG-PTR. See ARexx manual for an explanation.

RX.PUT.MSG (0arg 0portname -- error?)

Sends the argument to the named port. Uses NUL terminated strings. Converts the argument to an official ARexx ArgString and calls SafePutToPort(). RX.PUT.MSG then waits for and processes messages using RX.EXEC.MSG until the original packet is returned.

RX.PUT.REXX (0arg -- error?)

Send message to resident process REXX Calls RX.PUT.MSG with a portname of 0" REXX".

RX.PUT.TEXTRA (0arg -- error?)

Calls RX.PUT.MSG with a portname of 0" TEXTRA". The RXCOMM, RXFF_RESULT, and RXFF_STRING bits are first set in the action field then restored to just RXCOMM.

RX.REPLY.MSG (rexxmsg --)

Replies to message. Passes RX-RESULT1 (integer) and RX-RESULT2 (argstring) if they are set and the sender requested a reply by setting the RXFF_RESULT flag.

RX.SLAVE (-- , process ARexx commands until RX-QUIT on)

Wait for messages, execute them using RX.EXEC.MSG, then reply to them. This continues until the variable RX-QUIT is set to TRUE. You must provide a quit command to do that.

RX.SLAVE.SAFE (--)

Like RX.SLAVE but also quits if you hit a key.

RX.TERM (-- , terminate if initialized)

RX.WAIT.MSG (-- rexxmsg)

Be sure to reply using RX.REPLY.MSG when done with message. As an alternative to waiting for the ARexx message, you could AND several Signals together and use Wait(). This would allow you to get IDCMP events while waiting for ARexx events.

SafePutToPort() (msg 0portname -- ok?)

Finds a port with the given name then sends it a message. Uses Disable() and Enable() calls.

Integrating Textra and JForth

Textra is a text editor written in JForth by Mike Haas. It is optimized for program development as opposed to word processing. Textra is provided with JForth and can be found in the JTX: directory. Textra can be freely redistributed.

Mike has added an ARexx interface to Textra that allows you to manipulate text from other programs. Textra also provides the ability to compile programs directly from the Textra window without going to disk. If an error is encountered during compilation, the offending line is highlighted.

To prepare the Textra/ARexx interface, you will need to copy the scripts to the REXX: directory. Enter in the shell:

```
COPY JTX:SCRIPTS REXX: ALL CLONE
```

Now run Textra, select "AREXX..." from the Utilities menu. Click in one of the long text gadgets and enter "jcompile". We recommend the gadget associated with <F10>. Then click on the <SAVE> button. Finally, click the <CANCEL> button to close the requester.

To compile the Textra Interface into JForth, enter in JForth:

```
INCLUDE JRX:RexxClude.f
```

To activate the interface, enter:

```
RX.INSTALL
```

JForth will now be ready for any ARexx message sent to the port named "JFORTH".

> Note: Do not INCLUDE directly from Textra when you are writing programs that use the RX.xxx words. They will fight over the ARexx port and both will lose. Specifically, do not call RX.INIT when RX.INSTALL is active. Instead, write the file to disk and call the normal INCLUDE.

To test this interface, open a file using textra and enter a few lines of code. Put a deliberate error in one of the lines. The hit the <F10> function key.

JForth will begin to compile the contents of the Textra window. When it reaches the bug, it will highlight it in Textra. You can now fix the bug and hit <F10> again.

You can use SAVE-FORTH to save an image with this code already compiled. When you run this image RX.INSTALL will be called automatically by AUTO.INIT.

Warning: Do NOT select other windows or edit text while JForth is compiling from the window.

I recommend that you read the documents in JTX:DOCS. The file RexxCommand.doc contains a description of the commands that Textra supports. As long as we have Textra up and RX.INSTALL active let's have some fun.

Arrange your windows so that you can see the shell window, the JForth window and a Textra window all at once. In the Textra window, you should have a file with at least 20-30 lines for this experiment.

Note: Textra commands require a '@' before the command when sent from JForth. This is to distinguish commands from invocations of a script.

Lets try repositioning the Textra cursor. Enter in JForth:

```
0" @GOTOXY 10 5" RX.PUT.TEXTRA .
```

Did you notice the cursor jump in Textra. Enter these other commands and watch the cursor move.

```
0" @LEFT 3" RX.PUT.TEXTRA .
```

Now lets try selecting a line.

```
0" @SELECTLINE 8" RX.PUT.TEXTRA .
```

Now let's try retrieving the text that is currently selected in Textra. Select a different line if it is empty in your file. Enter:

```
: GETIT  ( -- , get and type )
   0" @get select text" rx.put.textra . cr
   rx-result2 @ 0count type cr
;
GETIT
```

The text is left at HERE which can be overwritten so be sure to copy it somewhere else if you want to keep it.

Now would be a good time to try out some of the commands described in the JTX:Docs/RexxCommand.doc file. You may also want to read some of the scripts in JTX:Scripts.

ARexx Variables Interface

The REXX Variables Interface (RVI) is a set of functions to allow an ARexx host to manipulate a macro program's symbol table. Using these functions the host can retrieve values for existing variables and install new values. There is no limit (except for available memory) to the number of variables that can be created, so the variables interface is a very convenient way to pass information to a macro program.

[Martin Kees created this code by disassembling REXXVARS.O which is a linkable object file supplied with ARexx. RVI was developed by William Hawes.]

The RVI functions can be called whenever the ARexx host holds a pending command message from an ARexx program. While the command message is outstanding, the macro program is waiting for the reply and the program's symbol table is in a consistent state. All calls to the interface functions must be completed before the message is replied.

In a typical application, a macro program issues a command to the host to perform some specific action, and may pass the name of a stem variable as an argument. The host performs the command and fills in the appropriate variables before replying to the message. When the macro program resumes operation, it can check the values of the variables set by the host.

Here is an outline of that transaction:

1) JForth host calls ARexx macro (optional).

2) ARexx macro sends command to JForth host.

3) JForth host can read or write macros variables using RVI.

4) JForth host replies to command.

5) ARexx macro can use new values in variables. (optional)

Variable Names

Variable names are specified by a pointer to a null-terminated string and must follow the REXX language symbol conventions. Alphabetic characters must be in UPPERCASE. The variable name is treated as a literal string, and no substitution for compound symbols is performed.

The host application should document the variables affected by each command so that the macro programmer knows where to find the information, and to avoid accidental conflicts in variable names. If a large number of values must be passed, the host should define them as compound variables and let the macro program pass the stem name as an argument.

The RVI Glossary

To use the RVI functions, include the file "JRX:RexxVars.f". It is not necessary to define the ARexx library base (RexxSysBase), as the ARexx library is opened on the fly when required.

CheckRexxMsg (rexxmsg -- ok?)

This function verifies that the message pointer is a valid RexxMsg and that it came from an ARexx macro program. The validation test is more stringent than that performed by the ARexx library function IsRexx(). The latter verifies that the message is tagged as a RexxMsg structure, but not that it necessarily came from an ARexx macro program. Each macro program installs a pointer to its global data structure in the command message, and this pointer is necessary to gain access to the symbol table.

The return from the function will be non-zero (TRUE) if the message is valid, and 0 (FALSE) otherwise.

GetRexxVar (rexxmsg 0variablename -- 0value error?)

This function retrieves the current value for the specified variable name. It first validates the message using CheckRexxMsg() and then, if the message pointer is valid, retrieves the value string and passes it in the supplied return pointer. The return pointer is actually an argstring (an offset pointer to a RexxArg structure), but can be treated as a pointer to a null-terminated string. The value must not be disturbed by the host.

The function return will be zero if the value was successfully retrieved and non-zero otherwise. An error code of 10 indicates an invalid message.

Here is an example of getting and displaying the value of a variable named PRICE.

```
MyRexxMsg 0" PRICE" GetRexxVar 0=
IF  ." Price = " 0count type cr
ELSE drop ." AN error occured getting PRICE" cr
THEN
```

SetRexxVar (rexxmsg 0variablename addrval length -- error?)

This function installs a value in the specified variable. It validates the message pointer using CheckRexxMsg() and then installs the value, creating a symbol table entry if required. The value is supplied as a pointer to a data area along with the total length; the data may contain arbitrary values and need not be null-terminated.

The function return will be zero if the call was successful and non-zero otherwise. The possible error codes are given in the table below.

Error Code Reason for Failure

```
3     Insufficient storage
9     String too long
10 Invalid message
```

Here is an example of setting a variable named PRICE to 1234.

```
MyRexxMsg 0" PRICE" 1234 n>text SetRexxVar
IF ." Could not set PRICE!" cr
THEN
```

Note that the value returned by GetRexxVar() is an argstring (pointer), and the value passed to SetRexxVar() is just a pointer to a data area.

RVI Code Examples and Test Program

A sample JForth program and an ARexx script to exercise the RVI functions are included. They are:

```
JRX:Tests/VarTest.f
JRX:Tests/TestRVI.rexx
```

VarTest opens a public port named "VarTest" and waits for messages to arrive from ARexx. Each message is validated with a call to CheckRexMsg(), after which the value for A.1 and A.2 is retrieved using GetRexxVar(). The program then installs the value "A-OK" in the variable STATUS and replies the message. VarTest exits when it receives a "CLOSE" command. The ARexx program TestRVI.rexx will demonstrate the VarTest host.

Note how the variable values are affected by the procedure calls in ARexx. In the first call from TestMem, A.1 will be NULL. Then it will be set to "Local to TestMem" for the next call. Then it reverts to its original value when we return from the procedure TestMem. This is because PROCEDURE in ARexx creates a new symbol table. A.2 keeps its value because it is EXPOSEd.

Here is the procedure for running this test program:

In JForth:

```
INCLUDE JRX:RexxVars.f  \ takes a while to assemble
INCLUDE JRX:tests/VarTest.f
VARTEST
```

In Amiga Shell:

```
RX JRX:Tests/TestRVI
```

RexxView

RexxView is a CLI utility that monitors the REXX port. Information is listed to a file describing each message received by REXX. JForth source code is in file jrx:RexxView.f. RexxView is designed to be cloned.

Usage:

```
rexxview  outfile
```

For example:

```
rexxview con:0/0/640/100/rv
(or)
rexxview PRT:
```

Rexxview is terminated by sending the string "closerexxview" to the REXX port. To do this enter in the CLI:

```
rx closerexxview
```

Chapter 24
Miscellaneous Amiga Support

What is supported and why?

You may be wondering how we decided which Amiga functions we would define JForth words for and which not. Our primary goal is to provide the necessary tools for accessing any Amiga functions. The CALL system and the Structure referencing tools provide this. These are theoretically all you need to access the Amiga libraries. This is why some functions are not directly supported. As for why some function are supported, there are four main possible reasons. The first reason might be that a function is so important that almost everyone will want it. The second is that the function might be a little tricky to implement so we will save you the pain, like the polygon or IFF code. The third reason is if it is a function normally present in 'C' but not part of the Amiga libraries, eg. CreateStdIO(). The final reason would be if we needed a function to write one of the demos or applications and then included it in a support file for all to use.

The routines that follow all pass addresses as relative addresses. Words that end in a '()' suffix map are simply calls to the Amiga library functions. Most of these calls are described in detail in the ROM Kernal Manual.

File JU:GRAPH_SUPPORT

These routines support various Graphics and Intuition Library calls.

ALLOC.BITMAP (bdepth bwidth bheight -- bitmap | false)

Allocate a BitMap structure with bitplanes based on the input. Calls InitBitMap(). This BitMap must eventually be freed using FREE.BITMAP.

ALLOC.RASTPORT (-- rastport | 0 , allocate and initialize)

ALLOC.SHADOW (bdepth bwidth bheight -- bitmap | false)

Allocates a special kind of bitmap that has only one plane allocated for each plane position. This is used for building shadow masks for transparent Blit operations. Must be freed using FREE.SHADOW.

ALLOCRASTER() (x y -- address)

BITMAP>WH (bitmap -- width height , extract information)

BltBitMapRastPort() (bitmap x y rp x y w h minterm --)

BltClear() (memblock bytecount flags --)

BMPLANE[] (index bitmap -- &plane-ptr)

This points to the position in the BitMap that holds the address of the plane. It is not the plane address itself. Do a @ >REL to get the plane address.Used when allocating and assigning planes both.

CLEAR.BITMAP (bitmap -- , set planes to zero)

Clear all the bit planes in a BitMap to zero using BltClear().

CLEAR.BITPLANE (plane# bitmap -- , clear plane of bitmap)

ClipBlit() (src_rp x y dest_rp x y w h minterm --)

CLOSEFONT() (font --)

COPY.PLANES (src_bm dest_bm -- , copy planes into other bitmap)

This copies the pointers to the planes of one BitMap to another BitMap. This can be useful for double buffering. Be careful you don't free these planes twice!

CTABLE>RGB (index ctable -- r g b , extract colors)

Extract Red, Green, and Blue values from a colortable. A colortable is packed 0RGB in 16 bits.

FREE.BITMAP (bmap -- , free planes then the Bitmap)

FREE.SHADOW (bmap -- , free plane then the bitmap)

For freeing Shadow mask allocated by ALLOC.SHADOW.

FREE.VIEW (view -- , free CPR lists and deallocate)

FreeCprList() (CPRList --)

FREERASTER() (address x y --)

FreeVPortCopLists() (viewport --)

INITBITMAP() (bitmap depth width height --)

INITRASTPORT() (RastPort --)

INITTMPRAS() (tmpras buffer size --)

INITVIEW() (view --)

INITVPORT() (viewport --)

LOADRGB4() (viewport colortable n -- , load color palette)

LOADVIEW() (view --)

MAKEVPORT() (view viewport --)

MRGCOP() (view --)

PACK.RGB (r g b -- ctable-entry)

Pack the 4 bit Red, Green, and Blue values into 16 bits organized as 0RGB.

RASSIZE() (height width -- size_in_bytes)

REMAKE.SCREEN (screen -- , rebuild Amiga display system)

RGB>CTABLE (r g b index ctable -- , store colors)

Opposite of CTABLE>RGB. Uses PACK.RGB

ROTATE.PLANES (bitmap -- , rotate planes for effect)

An interesting color shifting effect can be obtained if you rotate the order of the plane pointers in a Bitmap. This is very fast since you are only changing pointers. You must rotate as many times as there are planes to get back to normal.

SCALE.COLOR (color scalar -- color' , 0-16 scale)

Scale a 4 bit color value where the scale ranges from 0 to 16 where 16 is equivalent to 1.0.

```
7 16 scale.color  ( get 7 back )
7 8  scale.color  ( get 3 back )
```

SCALE.CTABLE (ctable1 ctable2 many scalar -- , scale c1 into c2)

Copy a colortable into another by unpacking the RGB values, scaling them, then repacking them back into the new colortable. Useful for fading in and out a picture.

SCALE.RGB (r g b scalar -- r' g' b')

SetRast() (rastport pen --)

SWAP.PLANES (bmap1 bmap2 -- , swap planes between bitmaps)

SWITCH.SCREEN (rastport screen -- , switch double buffer)

If the RastPort is used for drawing and the screen for displaying, you can create a double buffered image by drawing, then calling SWITCH.SCREEN.

WAIT.FRAMES (N -- , wait for N blanking intervals)

Loop on WaitTOF()

WaitTOF() (-- , wait till next vertical blank)

Font Support

See the demo JD:DEMO_FONT for an example of the use of these routines.

GET.FONT (0name ysize -- font | NULL)

Fills a text attribute structure with the necessary data and gets the disk based font. This font can then be passed to GR.FONT! for use with the graphics toolkit.

OPENDISKFONT() (textattr -- font | NULL)

OPENFONT() (textattr -- font | NULL)

SETFONT() (rp font --)

File JU:GADGET_SUPPORT

The Amiga Gadget system provides an flexible way to control your programs. We have provided a set of routines for initializing the common Gadget structures to typical values. You can then modify the values to suit your particular application. See the demo JU:DEMO_GADGET for an example of their use.

BOOLEAN.SETUP (x y width height gadget --)

BORDER.SETUP (width height border --)

BOX.SETUP (width height addr -- , set 5 points for box)

Used when creating a simple Border.

CHECKBOX.SETUP (x y width height gadget --)

INTGADGET.SETUP (x y width height gadget --)

ITEXT.SETUP (text0 intuitext -- , Set defaults.)

MENUBUTTON.SETUP (x y width height gadget --)

PROPGADGET.SETUP (x y width height gadget --)

PROPINFO.SETUP (hpot vpot hbody vbody propinfo --)

REFRESHGADGETS() (gadgets ptr req --)

STRINGGADGET.SETUP (x y width height gadget --)

STRINGINFO.SETUP (buffer maxchars stringinfo --)

File JU:POLYGON for Area Fill

To draw polygons you must first set up a temporary RastPort for creating a mask. GR.AREA.INIT will create one based on the current RastPort whose absolute address is in GR-CURRPORT. Then you can start a polygon by calling GR.AREA.MOVE followed by some number of calls to GR.AREA.DRAW then finally calling GR.AREA.END to draw the polygon. When you are all done drawing all your polygons, please call GR.AREA.TERM to deallocate the temporary RastPort. The system is set up with a limit of 100 points per polygon. Change the source code if you need more.

GR.AREA.INIT (-- , setup Temporary RastPort for polygons)

GR.AREA.TERM (-- , deallocate temporary RastPort)

GR.AREADRAW (x y --)

GR.AREAEND (--)

GR.AREAMOVE (x y --)

GR.TRIANGLE (x1 y1 x2 y2 x3 y3 --)

INITAREA() (areainfo areabuffer count --)

File JU:SCREEN_SUPPORT

The file JD:DEMO_HAM is a good example of these calls. Please see the Intuition manual for a detailed description of these calls.

BITMAP>SCREEN (bitmap viewmode -- screen | NULL)

Opens a screen that uses the given bitmap as its CUSTOMBITMAP.

CLOSESCREEN() (screen --)

MAKESCREEN() (screen -- , similar to MakeVport)

MOVESCREEN() (screen deltax deltay --)

A positive deltay will move the screen down. Deltax is ignored.

NEWSCREEN.SETUP (NewScreen -- , Set default values)

You can call this routine first then override the default values by setting them to your own.

OPENSCREEN() (NewScreen -- screen)

REMAKEDISPLAY() (-- , remake screens and COPR list)

RETHINKDISPLAY() (-- , reconstruct COPR list): SCREEN>VIEW (ascreen -- aview | NULL)

Moves screen to front then allocates a view based on the current display. This view can be used with LoadView() to create double buffered displays. Returns NULL (0) if the view could not be allocated. Free the view using FREE.VIEW

SCREEN>BACKWINDOW (screen -- window | NULL)

Creates a BACKDROP window to fit a screen. Sets the FLAGS and IDCMPFLAGS to the values in S>B_FLAGS and S>B_IDCMP_FLAGS. The default values are shown below.

```
BACKDROP BORDERLESS | WINDOWCLOSE | REPORTMOUSE |
-> S>B_FLAGS

CLOSEWINDOW MOUSEBUTTONS |
VANILLAKEY | MENUPICK | GADGETUP | GADGETDOWN |
```

```
        -> S>B_IDCMPFLAGS
```

```
SCREENTOBACK() ( screen -- , push to back )

SCREENTOFRONT() ( screen -- , bring to front )

SHOWTITLE() ( screen flag -- , SHow title bar or not.)
```

Exec Library Support

These words support most of the calls in the Exec Library. They can be found in the file JU:EXEC_SUPPORT. All addresses are passed as JForth relative addresses. Please see the ROM Kernal documentation for a description of these calls.

```
ADDPORT() ( port -- )

ALLOCSIGNAL() ( requested_signal# -- allocated_signal# )

CHECKIO()  ( ioreqblk -- ioreqblk | 0 )

CLOSEDEVICE() ( IORequest -- )

DOIO() ( ioreqblk -- error , Do I/O )

FINDPORT() ( 0$portname -- task )
```

The portname is passed as a NUL terminated string which must be generated using 0" inside of a colon definition.

```
FINDTASK() ( 0$taskname -- task )

FREESIGNAL() ( signal# -- , Free previously allocated signal. )

GETMSG() ( port -- message , Get message. )

OPENDEVICE() ( 0$devicename unit# IORequest flags -- error )

PUTMSG() ( port message -- , Put message. )

REMPORT() ( port -- )

ReplyMsg() ( message -- )

SENDIO()  ( ioreqblk -- error )

WAITIO()  ( ioreqblk -- error )

WAITPORT() ( port -- message )
```

Other Exec words - CreatePort() AbortIO() etc.

The following words are commonly used but are not part of the Exec Library. They are traditionally supplied as 'C' macros or are listed as 'C' source code in varous texts. They have been converted to JForth code for your convenience. You should always check the return value to make sure it is non-zero. Unless otherwise specified, these routines can be found in JU:EXEC_SUPPORT.

```
ABORTIO()  ( IOReqBlock -- result , abort an IO request )
```

This is a special call directly off of a Device structure. It returns a result but it is meaningless so you can drop it. We found out about this when it was too late to change it. This is defined in JU:DEVICE-CALLS. The above is also true of BeginIO().

```
BEGINIO()  ( IOReqBlock -- result , start an IO request )
```

See ABORTIO() definition.

```
CREATEEXTIO() ( ioreplyport size -- ioreq | 0 )
```

Allocate an extended IO Request Block that will use an existing IOReplyPort.

```
CREATEPORT()  ( 0$name priority -- msgport | 0 )
```

Allocate and add a new port to the system.

```
CREATESTDIO() ( ioreplyport -- ioreq | 0 )

DELETEEXTIO() ( ioext size -- )

DELETEPORT() ( port -- , Delete message port )

DELETESTDIO() ( ioreplyport -- )

PORT.SETUP ( 0$name priority signal msgport -- , Set values.)
```

Amiga Linked List Tools

These routines can be found in the file JU:EXEC_LISTS. They are Forth routines based on 'C' macros. They are handy for examining Amiga system lists, or for creating your own lists.

```
DUMP.LIST ( list -- , dump nodes of list )

DUMP.NODE  ( node -- , dump contents )

FORBID() ( -- , forbid other tasks )
```

If you are looking at system lists, you should forbid other tasks from changing them. Call PERMIT() when done.

```
LIST.ADDTAIL() ( list node -- , add node to end of list)

LIST.FIRST() ( list -- first_node , get first node in list )

LIST.IFEMPTY() ( list -- flag , true if empty )

NEWLIST() ( list -- , Initialize list header )
```

Set the pointers in a list header to point to itself. This signifies an empty list.

```
NODE.AFTER  ( node1 node0 -- , connect node1 after node0)

NODE.INIT  ( type priority 0name node -- , initialize node )

NODE.SUCC() ( node -- succeeding_node )

PERMIT() ( -- , permit other tasks after FORBID() )
```

ANSI Text and Cursor Control

These commands allow the control of text and the cursor in a console window. They send special control characters to the current output which are interpreted by the console device. These commands are used extensively by the Command Line History system. See also the file JU:COLORDUMP for more examples. Please see the ROM Kernal Manual for more information.

Note: The ANSI referred to here is the ANSI standard for terminal control characters, not the upcoming ANSI Forth standard. ANSI stands for American National Standards Institute. They standardize everything from languages to lightbulb sockets.

```
.DECIMAL  ( n -- , output decimal number without spaces )
```

This is used to generate formatted ANSI commands.

```
>STYLE  ( n -- , select graphic rendition )
```

This is used by ITALICS and other ANSI words to select a style.

```
ANSI.1C ( char -- , send CSI and one char )

ANSI.BACKWARDS ( N -- , move cursor backwards )

ANSI.DELETE ( N -- , delete character under cursor )

ANSI.DOWN ( N -- , move cursor down )

ANSI.ERASE.EOL ( -- , erase to end of line )

ANSI.FORWARDS ( N -- , move cursor forwards

ANSI.INSERT ( N -- , insert character )

ANSI.NC ( N char -- , send a number and a character sequence )
```

This is used to build special custom ANSI commands.

```
ANSI.PARSE.SKR  ( -- key-code, get packed key sequence)
```

If KEY returns a 155 decimal character, call this to parse the remaining characters. It will eat keys and return a FKEY index equal to the following. You can then use this in a CASE statement. Remember that if history is on it will eat these special keys so your program will never see them. Call HISTORY.OFF if you want to handle these characters yourself.

```
0 = error
1 = function key 1, 11 = shift 1,
2 = function key 2, 12 = shift 2, etc.
21 = cursor-up, 22 = cursor-down,
23 = cursor-right, 24 = cursor-left,
```

```
        25 = shift-cursor-up, 26,27,28 (shifted v < >)
        29 = help
```

ANSI.UP (N -- , move cursor up)

B:0 B:1 B:2 B:3 (-- , set text background color, see F:0)

BOLD (-- , make characters bold)

F:0 F:1 F:2 F:3 (-- , set text foreground color)

These words will change the text color. If history is interfering with this color, enter:

```
HIGHLIGHT-INPUT OFF
F:3  ( for fun )
B:2
```

GOTOXY (x y -- , move text cursor to column x, row y)

INVERSE (-- , style 7)

ITALIC (-- , style 3)

PLAIN (-- , style 0)

UNDERSCORE (-- , style 4)

Amiga DOS 2.0 Support

The ".j" include files were translated from Commodores AmigaDOS 2.0 ".h" files. The conversion from 'C' to Forth was done using H2J which is in JA:H2J.F

> JForth makes the full field of AmigaDOS 2.0 features available to the programmer. It is the programmer who must ensure that AmigaDOS functions and capabilities are not invoked under earlier versions of the operating system.
>
> CHECK **2.0FEATURES?** BEFORE CALLING AMIGADOS 2.0-SPECIFIC FEATURES AND FUNCTIONS!

Identifying the Workbench Version

2.0FEATURES? (-- flag)

Returns a TRUE flag if running under AmigaDOS 2.0 or later. The programmer should ensure his program is operating under 2.0 before using its services.

JU:ASL_SUPPORT

ASL - the Application Support Library

ASL is a new library provided in Amiga DOS 2.0. It provides facilities like file and font requesters. We urge programmers to use these requesters in their applications because they provide the user with a familiar interface. One problem, however, is that if the user is running AmigaDOS 1.3, these cannot be used. We suggest that you either require AmigaDOS 2.0 OR you should use the requester in JREQ:FileRequester.f when running under AmigaDOS 1.3.

Here are some direct calls to the ASL library. You must call ASL? before calling these routines.

AllocASLRequest() (type ptags -- request)

AllocFileRequest() (-- request)

ASLRequest() (request ptags -- result)

FreeASLRequest() (request --)

RequestFile() (request -- result)

ASL Forth Utilities

We have provided some words that use the ASL library as a convenience and as examples. These require AmigaDOS V2.0 or higher.

ASL.FILENAME>PAD (file-request -- , copy full name to pad)

This can be called after using the File Requester. It will concatenate the directory names and file names into a useable path name. The result will be left on the PAD. It is used internally by ASL.REQUEST.FILE

ASL.REQUEST.FILE (0prompt -- $name true | false)

Open the file requester with the given prompt. If the user selects a file, return its name and a true. If the user cancels the operation, return false. The filename will be left on the pad. See also the file requesters in JARP: and JREQ:.

```
: TEST.ASL ( -- )
   0" Select a file." ASL.REQUEST.FILE
   IF  COUNT TYPE CR
   THEN
;
```

ASL.REQUEST.FONT (0prompt -- font true | false)

Open the font requester with the given prompt. If the user selects a font, it opens it and returns a font pointer and a true. If the user cancels the operation, return false. This routine opens the diskfont library and calls OpenDiskFont(). When you are done with the font, you should pass it to CloseDiskFont(). You can use this font by calling Intuition's SetFont(). See also GR.FONT! in JU:AMIGA_GRAPH.

Tag Lists

TagLists are a new data structure in AmigaDOS 2.0. They are used to passed multiple attributes to a routine. The attribute identifier and a value are passed as pairs in a "list" that is terminated with a NUL. These routines simplify the creation and use of TagLists. Taglist support is in JU:TAGLIST.

TAGLIST{ (taglist --)

Start the definition of a taglists. You must decide where to put the taglist. You can put it in TAG-PAD, or in a memory area that you allocate.

}TAGLIST (-- taglist)

Finish the definition of a taglist. This returns the address you passed to TAGLIST{ as a convenience.

TAG-PAD (-- addr)

A place to put a taglist. The maximum number of tags it will hold is 16. Do NOT exceed 16 tags. If you need more tags, put them on the normal PAD or allocate memory. This is used by ASL.REQUEST.FILE.

TAG, (value -- , add to current taglist)

This is used by +TAG to add a single value to a taglist.

+TAG (attr value --)

Add an attribute/value pair to a taglist.

+TAG.ABS (attr rel-addr --)

Addresses in a taglist must be in absolute form for the Amiga. Use this word when adding addresses to a taglist. If will convert non-zero values to absolute using IF>ABS.

TAGLIST.DUMP (taglist -- , dump taglist maximum 100 long)

Dumps a taglist and shows each value as a number and as an address.

Here is an example of using these words to create and display a taglist. This taglist uses some of the tags for OpenWindowTagList().

```
INCLUDE? TAGLIST{  JU:TAGLIST
: MAKE.TAGLIST ( -- taglist )
   PAD       \ arbitrarily put it on PAD
   TAGLIST{  \ begin definition
      WA_HEIGHT  200  +TAG  \ set window height
\ now specify a custom screen ADDRESS
      WA_CUSTOM_SCREEN MY-SCREEN @ +TAG.ABS
   }TAGLIST
   dup taglist.dump
;
```

Key to Glossary

This glossary contains descriptions of the most common JForth words. To find words related to a specific function, eg. graphics or assembly, please look in the reference section.

The words are organized in ASCII alphabetical order. Many Forth words start with punctuation. You can find out the alphabetical order for punctuation by looking at the bottom of the page in the glossary.

Each Forth word description has a common format. The first line of the description has the name of the word followed by a stack diagram. The pronunciation of the word sometimes follows.

Following the description of the word will often be an example, indented and in upper case.

If the word is not in the normal JForth image, the file where it can be found will be given.

Any related words will then be listed at the end of the description. Please also see the appendix on words related by function as well.

Understanding Stack Diagrams

Stack diagrams tell you what parameters are passed to a Forth word and what is returned. The input and the output is separated by several dashes. Look at the stack diagram for + as an example.

```
+ ( a b -- a+b , add two numbers together )
```

This means that + takes two numbers, A and B, as input. It leaves A+B as output. The text after the comma is a comment. The rightmost item is on the top of the stack so B is on top in this example.

String Parameters

There are a number of ways to pass a string in JForth.

Parameters that start with a $ are Forth "counted strings". The parameter will be the address of a count byte that will be followed in memory by that number of characters. The Forth string "FROG", for example, will be stored in memory as:

```
" FROG" 5 DUMP  ( reveals 04 46 52 4F 47 )
```

The first byte is the count byte. The 46, 52, 4F, 47 are the ascii characters F R O and G.

Forth also supports NUL terminated strings like 'C'. Here, the address of the first character is passed. The string continues until a zero is encountered.

```
0" FROG" 5 DUMP ( reveals 46 52 4F 47 00 )
```

These strings parameters are referred to with a zero in the name, eg. 0string.

Forth strings can also be passed as an address and a count.

Please examine the stack diagrams for COUNT and 0COUNT as examples.

```
COUNT   ( $string -- addr count )

0COUNT  ( 0string -- addr count )
```

Here are examples of the use of each.

```
" FROG" COUNT TYPE
0" FROG" 0COUNT TYPE
```

Strings can also be passed on the input line. Consider the Forth word FOPEN which expects a filename to follow. FOPEN opens a filename of the given name and returns a file-pointer which can be used to read the file. Here is an example of a call to FOPEN .

```
FOPEN RAM:MYFILE
```

Strings that are passed on the input line are enclosed in angle brackets in stack diagrams. The stack diagram for FOPEN would, therefore, be:

```
FOPEN   ( <filename> -- file-pointer | false )
```

The vertical bar and the FALSE means that if the file cannot be opened, a FALSE will be returned as an error indicator.

Please keep in mind that some words may have an action but may not have any stack activity. The word CR, for example has a stack diagram of:

```
CR ( -- , emit carriage return )
```

Return Stack

There is also a Return Stack in Forth. Words that affect the Return Stack will have a Return Stack Diagram. These will be marked with a --R--. Here is an example.

```
>R   ( n -- , --R-- n , move N to the return stack )
```

Summary

In summary, here is a description of the common parameters you will see:

```
a b     = some values
addr    = address of some memory location
flag    = TRUE (-1) or FALSE (0)
char    = ASCII character
count   = number of things, usually bytes
d       = double precision integer, 64 bit
n       = single precision integer, 32 bit
pad     = address of PAD , a scratch area
u       = unsigned integer
var-addr = address of a variable
$addr   = counted string address
$string = counted string address
0string = NUL terminated string address
<text>  = text follows word
<name>  = name of a Forth word
<filename>      = name of a file
|       = OR , denotes alternate parameters
```

! (n addr --) "store"

```
addr = even address
```

Store 32 bit value N at even address ADDR. This is the primary word for storing data in memory. Variables and other JForth data structures will always be word-aligned, ie. have an even address. Code which may conceivably store 32-bit values to odd addresses should use ODD! . The address is assumed to be a JForth relative address, not a 68000 absolute address.

```
\ create and set a variable to '234'
VARIABLE VAR1
234 VAR1 !
```

Related Words: >REL >ABS ODDD! ODD! ODDW! CMOVE FILL ODDW@ D! W! C! +! @ C@ W@ ..@ ..! TRAPS

!CSP (--) "store C S P"

Stores the current parameter stack address in the user variable CSP. Used with ?CSP to see if parameter stack is balanced and there were no compilation errors. Used to check stack depth between : and ;

Definition: : !CSP (--) SP@ CSP ! ;

Related Words: CSP ?CSP

" (<string> -- $addr) "quote"

Returns the address of quote delimited string. The string will be stored as a count byte followed by the text. Note that a space is required after the first quote. If encountered while COMPILING, quote compiles the string into the dictionary, returning that address when executed. If encountered while INTERPRETING, quote places the count at PAD, and the string at PAD+1, returning the PAD address. Be aware that the PAD is used by many Forth words so strings generated when INTERPRETING are considered temporary.

```
" Hello" COUNT TYPE
```

Related Words: EMIT COUNT TYPE 0"

 # (d1 -- d2)

```
"sharp" ( fig '83 )  "number" ( '79 )
```

Used to convert numbers to a character string. Divide D1 by current BASE, leaving remainder as D2. Convert quotient to ASCII character, place character at address in HLD (below PAD), decrement HLD.

```
: PHONE#  ( N -- , Print N as phone number)
    S->D <# # # # #  ASCII - HOLD  #S #>
    ( -- addr count ) TYPE SPACE ;
```

Related Words: #S <# #> HOLD PAD HLD COMMAS NO-COMMAS SIGN

#> (d1 -- text_addr count)

Drop D1, terminating numeric output conversion, leaving address and count of the string.

Standards: '79, '83, FIG.

```
: UD.  ( d1 -- , print an unsigned double number)
    <#  #S  #>  TYPE ;
```

#DIGITS (N -- #characters) "number digits"

Calculates how many characters are needed to display a number N, including commas if enabled.

Related Words: N>TEXT # NO-COMMAS COMMAS

#K (-- addr) "number k"

Variable used by SAVE-FORTH to determine the size of a JForth image file. To increase the size of your dictionary, set this variable, do a SAVE-FORTH, then run the resultant image.

```
\ Increase dictionary size by 20K
20 #K +!
SAVE-FORTH MyImage
```

Related Words: MAP SAVE-FORTH #U

#RELOCS (-- addr) "number relocs"

A user-variable, containing the number of long absolute relocations that have been compiled. Internal.

Related Words: PUSHRELOC .IMAGE MAP

#S (ud -- 0 0) "number s"

#S converts the unsigned double value on the stack into an ASCII character string in memory. String location starts one byte below PAD and works down. Produces only enough digits to represent the number. No leading zeros. Always produces at least one digit which may be zero. See # and #> .

Related Words: <# # #> SIGN HOLD HLD PAD #S

#TIB (-- addr) "number t i b"

A user-variable containing the TOTAL number of characters in the TIB (terminal input buffer).

```
: DUMP.TIB    TIB #TIB @   TYPE ;
```

Related Words: TIB 'TIB >IN QUERY

#U (-- addr) "number u"

#U is a variable that contains the number of allowable user variables. If you run out of user variables, increase #U and do a SAVE-FORTH. Works like #K .

Warning: once you make #U larger, you can't make it smaller. You must go to an earlier Forth to get smaller user area.

#VOCS (-- addr) "number vocs"

A user-variable, containing the current number of vocabularies defined in the dictionary.

JForth allows a maximum of 32 vocabularies, two of which are defined in the minimum development system. #VOCS helps you to keep track of what vocabularies are available and what words can be used.

Related Words: #CHARS

$ (<hex-number> -- N) "dollar"

$ is used to force the interpretation of the next word in the input stream as a hexadecimal number, regardless of the current BASE. $ is an IMMEDIATE word and, if found within a colon definition, will also compile the value into the dictionary as a LITERAL.

```
decimal 256   $ 80 -   .  ( prints 128 )
```

Related Words: HEX DECIMAL

$" string quote See "

$, string comma

(char --) (<text> --in--) (--inline-- <text>)

Takes text from the input stream, delimited by char, compiles this text into the next available dictionary location.

$, is a primitive used by the string literal words. It performs the actual job of installing the text into the dictionary.

$, is usually only called from higher-level words, such as " .

$- ($1 $2 -- flag) "string minus"

Compare two strings alphabetically, return:

```
flag = 0  if strings equal,
flag = +1 if $1 greater than $2,
flag = -1 if $2 greater than $1.
```

Related Words: $= COMPARE

$= ($1 $2 -- flag) "string equals"

$= performs a case-sensitive compare between 2 strings and returns true if equal, else false.

For case-INsensitive string matches, see MATCH? and TEXT=? .

Related Words: $- COMPARE MATCH? TEXT=?

$>0 ($string --) "string to zero"

Convert a normal Forth string, with a count byte, to a 'C' type NUL terminated string. The conversion is done in place by shifting the characters down by one and placing a NUL, zero byte, at the end. NUL terminated strings are used when passing strings to the Amiga libraries.

Related Words: 0COUNT >DOS

$APPEND (addr count dest-string --) "string append"

This word is used to append text to the string at destination-string address.

The count at address destination-string will be updated to reflect the increased string size.

It is the responsibility of the calling program to insure that sufficient space exists above address destination-string to accommodate the increased string size.

$DOLINES ($filename --)

This is equivalent to DOLINES except it takes a filename on the stack. See DOLINES

$DOS ($doscommand --) "string dos"

Execute a string as a DOS command.

```
: PWD ( -- , Print working directory. )
    " cd" $DOS ;
```

$FOPEN ($filename -- file-pointer) "string f open"

Opens the file whose name is on the stack. $filename is the address of a count byte. A zero is returned if the file could not be opened.

See FOPEN, 0FOPEN and chapter on File I/O.

```
: OPENTEMP  ( -- file-pointer )
    " ram:temp" $FOPEN DUP 0=
    WARNING" RAM:TEMP could not be opened!" ;
```

$LOGTO ($filename --)

Log all output to the file whose name is on the stack. See LOGTO

$MOVE ($addr-source addr-dest --) "string move"

$MOVE moves a complete string, including the count byte.

```
VARIABLE MYSTR 256 ALLOT
" A text string" MYSTR $MOVE   ( move string to MYSTR )
```

$SIZE ($addr -- true-size) "string size"

$SIZE takes a string count-byte address and returns the number of bytes in the string plus the count byte plus any padding required to word-align the end. Use COUNT if you want the actual number of characters in the string, without count-byte and padding.

$TYPE ($addr --) "string type"

$TYPE is a shorthand for COUNT TYPE. It takes a string count-byte address and types out that string.

' (<word> -- cfa) "tick"

All words in the dictionary have a body containing executable 68000 code. This word returns that address for any entry in the CONTEXT vocabulary.

If executed within a colon definition. this IMMEDIATE word will locate the cfa of the next word as usual, then compile it as an address literal (ALITERAL) to be pushed to the stack at run-time.

```
\ Use ' for indirect execution.
: FOO ." Hello world!" CR ;
' FOO   EXECUTE   ( does FOO )
```

'>BODY (cfa -- pfa) "tick to body"

See >BODY

'>NAME (cfa -- nfa) "tick to name"

See >NAME

'TIB (-- addr) "tick T I B"

This is a user-variable containing the JForth relative address of that users TIB area. The value of this user-variable is placed on the stack by TIB .

'WORD (-- $addr) "tick word"

'WORD is a DEFERred word that returns the address that WORD will use to PLACE its output.

'WORD usually is set to execute HERE.

((<text> --) "left parenthesis"

Text between parentheses is considered to be a comment in Forth. The word (will eat text until the closing) .

The opening (must be followed by a space. The comment is terminated with the next) character, or end-of-line. Note that a comment may NOT span more than one line.

```
( for humans only , put anything you want inside )
```

Related Words: \ .IF

($") "parenthesis string quote"

Compile time: (--) (<string> --in--) (--inline-- <string>)

Run time: (-- $addr)

($") is used in inline string words like " $" etc... ($") is compiled before using $, to place a string inline. At run time, ($") will place the inline address of the string on the stack, and skip over the inline string to continue executing the code after it.

```
: $\ COMPILE ($") ASCII \ $,  ; IMMEDIATE
: FOO $\ A string with " in it!\ $TYPE ;
```

Related Words: $" 0" (($")) "

(($")) "double parenthesis string quote"

(-- $addr) ($addr ip --R-- ip)

(($")) is used in inline string words like (?ABORT") . (($")) is used within words that must parse inline strings. It is almost the same as ($") but it is used at 1 level of calling lower. See the STRINGS and STRING+ files for examples.

```
: (?ABORT")  (($"))  SWAP
    IF   CR $TYPE QUIT
    THEN  DROP ;
: ABORT"  COMPILE (?ABORT") ASCII "   $,  ; IMMEDIATE
```

((CREATE)) (--) "paren paren create"

This word creates a dictionary header out of a valid JForth string residing at HERE. The dictionary pointer DP is left pointing to the code-field-address cfa , with the name-field-address being left SMUDGEd.

((CREATE)) is a primitive used by all defining words .

(+LOOP) "paren plus loop"

(inc --) (n1 n2 --LOOP-STACK-- n1 n2+inc)

```
n2 = index approaching overflow
n1 = correction value to recreate index for I
inc = increment amount
```

(+LOOP) is the run time action compiled by +LOOP . Used with DO .

(+LOOP) adds inc to n2. If n2 overflows, the LOOP is satisfied, and execution continues once the loop indices have been dropped from the LOOP-STACK. If n2 does NOT overflow, return to the address immediately following the corresponding DO code.

Related Words: LOOP DO -LOOP +LOOP -DO LOOP-BACK LOOP-FORWARD REPEAT DO_FLAG

(-LOOP) "paren minus loop"

(inc --) (n1 n2 --LOOP STACK-- n1 n2-inc)

```
n2  = index approaching overflow
n1  = correction value to recreate index for I
inc = increment amount
```

(-LOOP) is compiled by -LOOP . Used with DO .

(-LOOP) subtracts inc from n2. If n2 overflows, the LOOP is satisfied, and execution continues once the loop indices have been dropped from the LOOP-STACK. If n2 does NOT overflow, return to the address immediately following the corresponding DO code.

Related Words: LOOP DO -LOOP +LOOP -DO LOOP-BACK LOOP-FORWARD REPEAT DO_FLAG

(.") "paren dot quote"

(--) ($addr --R-- addr)

```
$addr = string address
addr  =  address of next word to be interpreted
```

This word is a run time string-handler, used to print the string immediately following it's own compiled location. It also modifies the program counter to jump around the string. It is compiled by ." to output the string when executed.

(.) (n --) "paren dot"

The number primitive executed for signed integer output in the current BASE, usually executed by . (dot).

Definition: : (.) (n --) S->D D. ;

(;) (--) "paren semi-colon"

This word is a primitive used in closing any colon definition. It does the following.

1. Verify stack integrity (via ?CSP).
2. Compiles the RTS opcode (necessary at the end of all JForth definitions).
3 Reveals the name-field to FIND, so that it may accessed.
4. Terminates COMPILE mode, returning to INTERPRET mode.

(?DO) (limit index --) "paren question do"

(?DO) is compiled by DO .

At run time, (?DO) check to see if the difference between the index and limit is a positive, non-zero value.

- If so, it converts the index and limit to LOOP parameters, and pushes them on the LOOP-STACK.
- IF not, it drops both parameters, and jumps past the corresponding LOOP word.

(?LEAVE) (flag --) "paren question leave"

(?LEAVE) is compiled by ?LEAVE . Used for conditionally leaving a DO LOOP.

(?LEAVE), at run time, will check a flag on the stack...

- If non-zero, it will drop the loop indices and jump past the corresponding LOOP word.
- If zero, operation continues.

(?TERMINAL) (-- true-if-key-flag) "paren question terminal"

(?TERMINAL) is the run time code for ?TERMINAL . ?TERMINAL is a deferred word normally set to (?TERMINAL) . In the '83 standard this is referred to as (?KEY) .

(ABORT) (--) "paren abort"

This is the default word which the DEFERred word ABORT executes. It calls QUIT.

(CFA,) (cfa --) "paren c f a comma"

Compile the appropriate code necessary to execute the word whose CFA is on the stack.

Once the compiler has decided to compile a reference to an existing word, this is the primitive which figures out the particular manner of compilation which will be used, ie. INLINE, BSR, short or long JSR.

This is the default contents of the DEFERred word CFA, and is the real workhorse of the compile process.

(CFA,) determines whether the referenced CFA is of the INLINE type. If so, a copy of the body of that definition is moved to the current DP, and its length is ALLOTed from the dictionary.

If (CFA,) determines the word is of the BOTH variety, it checks the value of MAX-INLINE, and if the length of the word is less, it is compiled inline; otherwise referenced via a call, as shown.

! " # $ % & ` ( ) * + ' - . / 0-9 : ; < = > ? @ AZ [\ ] ^ _ az { | } ~

If the word must be referenced via a call, the compiler will select the most efficient of 3 types of subroutine calls to use.

If the called word is in the lowest 96K of the dictionary, the compiler will use the JUMP-SUBROUTINE (JSR), INDEXED WITH DISPLACEMENT mode, either through register A4 (ORG) or A3 (+64K).

If the called word is not in the lower 96K of JForth, but is within 32K of the calling instruction, the BRANCH-SUBROUTINE (BSR) mode is used.

Otherwise, the compiler will use the JUMP-SUBROUTINE (JSR), LONG ABSOLUTE mode. This type of reference requires the JForth System Manager to maintain a table of all long absolute references, that they may be relocated by the Amiga Loader at startup. This table is invisible to the programmer.

(COMMAS) (-- addr) "paren commas"

This is the user variable that the number-formatting and printing routines check to see if they should include commas in the output stream. If true, include commas.

Related Words: COMMAS NO-COMMAS

(CR) (--) "paren carriage return"

(CR) is the default contents of the DEFERed word CR . (CR) outputs a carriage return and a line feed to the standard EMIT device.

(CREATE) (<name> --) "paren create"

(CREATE) is the default contents for the DEFERed word :CREATE .

(CREATE) eats the next word in the input stream (parsed by the character value stored in CREATECHAR), and from it, creates a new dictionary header, in the CURRENT vocabulary. The name is left smudged.

Related Words: :CREATE REDEF?

(DO) (limit index --) "paren do"

(DO) performs similarly to (?DO), except that it performs no index/limit value checking. This means that: 1. If the index and limit are equal, the LOOP will be executed once. 2. If the index is greater than the limit, it will LOOP until the index has been incremented and is subsequently equal to the limit.

NOTE: this word, when used with LOOP, may be used for positive-growing LOOPS:

(EMIT) (char --) "paren emit"

This is an EMIT function primitive that performs single-character I/O to the CONSOLE window. It is executed by the deferred word EMIT when in the SLOW I/O mode.

This word transfers characters to the screen in single-character fashion. The normal I/O technique utilized in JForth is buffered, and this word is only used, oddly enough, for KEY, to echo each character typed to the user. In the SLOW I/O mode, it is the only EMIT primitive used.

(EMIT) will update OUT to reflect the current cursor column number.

NOTE: (EMIT) is NOT the default contents for the DEFERed word EMIT. See <FASTEMIT>.

(EXPECT) (addr nchars --) "paren expect"

Default word for EXPECT . See EXPECT . EXPECT is deferred so that it can be changed by the user if needed.

(FIND) "paren find"

($name lfa -- cfa true | $name false)

(FIND) is a primitive used to locate the most recently defined occurrence of the given name. It will start searching the dictionary at the lfa passed on the stack.

If (FIND) locates the name, it returns its CFA and a TRUE flag.

If the word is not found, (FIND) will return the original name and a FALSE flag.

(FIRST) (-- addr) "paren first"

This is a user-variable that is only used in the BLOCK environment, and is initialized when the source file JU:BLOCK is compiled.

After initialization, it contains the address of the first virtual buffer available under BLOCK.

(ID.) (nfa --) "paren i d dot"

This is the default contents of the DEFERred word ID. . It accepts the count byte address of a name-field, and TYPEs the name on the standard EMIT device.

(INTERPRET) (--) "paren interpret"

Run time code for INTERPRET . This is the default contents of the DEFERed word INTERPRET. It is the core of what is known in Forth as the "Outer Interpreter". JForth does not have an "Inner Interpreter since it compiles directly to 68000 machine code. (INTERPRET) is what interprets and executes the commands that you enter at the keyboard. It is also used when compiling a file.

(INTERPRET) gets its input from the TIB. (INTERPRET) uses >IN to index into the TIB. When >IN equals #TIB, or an error occurs, (INTERPRET) returns.

After (INTERPRET) has parsed a word from the input stream, it searches first the CONTEXT, then CURRENT vocabularies. If found, the resultant address will either be executed or compiled, based on the STATE of the system, and (INTERPRET) will continue with the next word.

If not found, it will see if the text can be successfully converted to a number (using the current BASE) and if so, the resultant number is passed to LITERAL.

If a number cannot be derived, the system QUITS, displaying the offending text on the console.

Related Words: QUERY $INTERPRET

(KEY) (-- char) "paren key"

Wait for a character from the console. When received, return the character without echo. Flushes any pending output. Default contents of DEFERred word KEY .

(LEAVE) "paren leave"

(--) (n1 n2 --LOOP-STACK--) n1 and n2 are loop indices

(LEAVE) is compiled by LEAVE.

(LEAVE) forces immediate termination of a DO LOOP by dropping the loop indices, and jumping past the corresponding LOOP word.

(LIMIT) (-- addr) "bracket limit"

This is a user-variable that is only used in the BLOCK environment, and is initialized when the source file JU:BLOCK is compiled.

After initialization, it contains the address of the END of the virtual buffer area available under BLOCK.

(LONGCFA,) (cfa --) "paren long c f a comma"

This is the default contents for the DEFERred word LONGCFA, .

This word is used to compile a LONG ABSOLUTE JUMP-SUBROUTINE to a cfa in the dictionary. (LONGCFA,) will automatically create an entry in the Long Relocations table for that reference.

(LOOP) paren loop

(--) (n1 n2 --LOOP-STACK-- n1 n2+1)

```
n2 = index approaching overflow
n1 = correction value to recreate index for I
```

(LOOP) is compiled by LOOP.

(LOOP) increments n2 by 1. If n2 overflows, the LOOP is satisfied, and execution continues once the loop indices have been dropped from the LOOP-STACK. If n2 does not overflow, return to the address immediately following the corresponding DO code.

(NUMBER) ($string -- d true | false) "paren number"

Default contents of deferred word NUMBER . This converts the character string at $string to a double number using the current BASE. If the conversion fails, false is returned.

(OF) "parenthesis of"

(case-value of-value -- case-value)

(OF) is the internal run time function for OF used in CASE statements. It compares the top item on the stack with a duplicate of the next thing on the stack. If they are equal it does not branch, else it branches to the ELSE part of the OF ... ELSE structure.

(QUIT) (--) "paren quit"

(QUIT) is used to terminate the currently running program and initialize the INTERPRETer to return to the CONSOLE. When called, it has the following specific effects:

```
1. Closes all files that had MARKFCLOSE executed on them.
2. Frees all memory blocks that had MARKFREEBLOCK executed on
them.
```

```
3. Initializes the TIB from TIB0, #TIMES variable to 1.
4. ALIGNS the dictionary.
5. Terminates COMPILE MODE, forcing INTERPRET MODE.
6. Clears the USP stack.
7. Enters the QUERY INTERPRET loop.
```

This is normally executed by QUIT and performs many JForth-System initializations. If you desire to change the operation of QUIT, you should only supplement (QUIT), executing your code then calling (QUIT) . When the JForth Kernal is TURNKEYed, the action of (QUIT) MUST BE REPLACED with the startup word for your turnkey operation.

(QUIT) is the default contents of the DEFERed execution word, QUIT, and may remain so in any version of your application that has not been distributed or released in any way. You are required to replace this cfa with one of your own definition in a TURNKEYed application. See TURNKEY and CLONE.

(SOURCE) (-- TIB #TIB) "paren source"

```
TIB  = Terminal Input Buffer address
#TIB = Number of characters in TIB
```

This primitive returns the address and length of the TIB.

*** (n1 n2 -- n1*n2) "times"**

Multiply two 32 bit numbers.

```
\ Multiply 23 time 7
23 7 * .  ( prints 161 )
```

***/ (n multiplier divisor -- n*m/d) "times slash"**

*/ calculates n times multiplier as a double length value. This is useful if the product would be greater than 32 bits.

```
1,000,000,000 234 567 */  ( correct )
1,000,000,000 234 * 567 / ( overflows! )
```

***/MOD (n1 n2 n3 -- n4 n5) "times slash mod"**

```
n5 = n1 times n2 and then divided by n3
n4 = remainder
```

*/MOD carries n1 times n2 as a double length value. This allows greater accuracy than would a single length intermediate product. Then n3 is divided into n1 * n2 yielding n5 quotient and n4 remainder.

```
2 5 7 */MOD  ( yields 3 1 )
```

+ (n2 n1 -- n1+n2) "plus"

+ adds the top two values on the stack and places the result on the stack.

```
3 4 + .   ( prints 7
```

+! (n addr --) "plus store"

+! adds the value n to the value stored at address addr and leaves the result at addr .

```
VARIABLE VAR1
```

```
10 VAR1 !
5 VAR1 +!
VAR1  ?   ( prints 15 )
```

+- **"plus minus"**

```
( n1  n2  --  n1 )    if n2 > 0 or n2 = 0
( n1  n2  -- -n1 )    if n2 < 0
```

+- negates the sign of second stack value n1 if top of stack value n2 is negative. Top value n2 is then dropped.

```
5 -7 +- .   ( prints -5 )
```

+DOS (addr count --) "plus dos"

Adds a string to the existing null-terminated string at DOS0 . This is used when interfacing with Amiga DOS.

```
" df0:c/" COUNT >DOS
" dir" COUNT +DOS
DOS0 0COUNT TYPE  ( print "df0:c/dir" )
```

+LOOP "plus loop"

Compile time: (loop-addr do-flag --)

Run time: (inc-val --) (index limit --R-- index limit |)

Used in DO LOOPs where you want to increase by some number other than 1.

+LOOP checks for a do-flag at compile time and gives error if not found. +LOOP compiles (+LOOP) At run time (+LOOP) increments index by inc-val and jumps back to loop body until index crosses boundary between limit and limit-1 (overflows).

```
: TDO+ 50 0 DO I . 10 +LOOP ;
TDO+   ( print 0 10 20 30 40 )
```

+SHIFT (a b -- a<<b) "plus shift"

+SHIFT is a faster smaller routine than SHIFT that does only left shifts toward the MSB. 0's are shifted into the LSB positions.

```
3 2 +SHIFT .  ( prints 12 )
```

Related Words: SHIFT ASHIFT -SHIFT U2*

+STACK (n var-addr --) "plus stack"

+STACK is used to PUSH items to a memory block (allocated via ALLOCBLOCK) being used as a stack. It accepts as arguments, the value n1 to push and the address var-addr of a VARIABLE or USER-variable being used to hold the address of the allocated memory area.

If the VARIABLE or USER-variable contains 0, +STACK will allocate a 1024-byte block to be used, sufficient for 256 cells, and place the address of the block in the storage location. It will then push the value to the area.

If the VARIABLE or USER-variable contains other than zero, it is assumed by +STACK to be the address of an existing memory block.

If the allocated stack area is full, and cannot accept the new element, +STACK will attempt to increase the allocated size in 1K increments.

Please see the chapter on Memory Management.

Related Words: PUSH POP -STACK

! " # $ % & ` ( ) * + ' - . / 0-9 : ; < = > ? @ AZ [\ ] ^ _ az { | } ~

, (n --) "comma"

'Comma' compiles N into the dictionary at HERE and advances the dictionary pointer by 4. It can be used to supplement CREATEd data structures in the dictionary.

```
\ Create an array called sizes
CREATE SIZES 330 , 250 ,  230 , 470 ,
2 CELLS SIZES + @ .  ( prints 230 )
```

Related Words: ALLOT HERE ! @ W,

- (a b -- a-b) "minus"

Subtract top of stack B from next to top A. Result is put on top of stack.

```
10 7 - .  ( prints 3 )
```

Related Words: + CELL- 2- 1- D-

-2SORT (n1 n2 -- biggest smallest) "dash 2 sort"

Sort top two items on stack into reverse order.

```
: SAFEDO  -2SORT DO ." hello" cr LOOP ;
10 0 SAFEDO
0 10 SAFEDO  ( both work )
```

Related Words: 2SORT MIN MAX

-> (n <name> --) "to"

Store N into a value data structure or a local variable. VALUE data structures and local variables are self fetching so you can't use Reverse Polish words like ! . See VALUE for example.

File: JU:VALUE or JU:LOCALS

-CLIST -CONSOLE -DISKFONT -GRAPHICS -ICON -INTUITION -LAYERS

Close the appropriate Amiga Library IF open. See LIBRARY . This should be done before terminating an application on each library that had been opened.

-DO "minus do"

(to from --) (--R-- previous-loop-parameters)

Begin a DO LOOP without checking the to and from values.

It is often used when you want to count down instead of up, in a loop, and is normally used with -LOOP as a result. You may also use -DO if you want to save space and time in setting up ordinary loops, where you do not need the limit checking of DO.

The TO an FROM values are processed at run time into limit-overflow values and placed in 2 loop parameter data registers.

```
\ Countdown from 10 to 0. Note 11 loops.
: COUNTDOWN   0 10 -DO I . 1 -LOOP ;
```

Note that the following loop with -DO will execute once. The equivalent loop with DO will not execute.

```
: ONCE  0 0 -DO ." Hello" cr LOOP ;
```

Related Words: DO LOOP -LOOP +LOOP

-DOS (--) "minus dos"

This word has no effect in JForth. Two libraries, EXEC and DOS are opened by the JForth startup routines, and should only be closed by BYE. It is provided to be compatible with the word sets provided for the other library definitions.

-DUP (n1 -- n1 n1 | 0) "dash dup"

Duplicates the top of the stack only if it is non zero. This can save you from having to do ELSE DROP.

Standards: '79 and '83 use ?DUP for this definition. FIG uses -DUP

```
: MAYBECLOSE  ( filepointer -- , close if safe )
    -DUP IF FCLOSE THEN ;
```

Related Words: ?DUP DUP XDUP IF 0=

-EXEC (--) "minus exec"

This word has no effect in JForth. Two libraries, EXEC and DOS are opened by the JForth startup routines, and should only be closed by BYE. It is provided to be compatible with the word sets provided for the other library definitions.

-FIND (<word> -- pfa cntadr true | 0) "dash find"

FIG standard version of FIND. Not recommended.

File: JU:MULTISTANDARD

-LIB (var-holding-lib-pointer --) "minus lib"

This word is the primitive used by all the library closing routines.

It accepts the address of a VARIABLE or USER-variable that contains the pointer to an open library. If the var contains 0, -LIB does nothing. The word set provided for handling Amiga libraries usually precludes the necessity for the programmer to use -LIB.

Related Words: LIBRARY

-LOOP (n --) "minus loop"

-LOOP works like +LOOP but subtracts the value on the stack instead of adding it. -LOOP is normally used with -DO for loops where the count is counting down instead of up. See -DO

Related Words: -DO +LOOP DO LOOP (-LOOP) (+LOOP)

-MATHFFP -MATHIEEEDOUBBAS -MATHIEEESINGBAS -MATHTRANS -POTGO

(--)

These words will close the appropriate Amiga library, if it is open. If not, there is no effect.

Related Words: LIBRARY

-ROT (a b c -- c a b) "minus rot"

Rotate the top three items on the stack in the reverse direction that ROT would. Equivalent to but faster than using ROT twice.

-SHIFT (n s -- n>>s) "minus shift"

Shift n by s positions to the right.

-SHIFT is a shorter, faster version of SHIFT that only does negative shifts toward the LSB. See SHIFT for more info. Zeros are shifted into the MSB so be careful with negative numbers.

```
100 2 -shift . ( prints 25 )
```

Related Words: SHIFT +SHIFT ASHIFT U2/

-STACK (value var-address --) "minus stack"

```
value           = item that is searched for and removed
var-address     = address of a variable
```

-STACK is used to remove items from a memory block that is being used as a stack. It expects the address of a VARIABLE or USER-variable that points to the allocated memory area, and the value to find and remove. If the VARIABLE or USER-variable contains 0, nothing will happen. Otherwise, the stack area is searched for the value, removed if found, and the FREEBYTE counter for the area updated. If -STACK causes the stack area to become empty, it will free the memory area, clearing the pointer address that was passed as an argument. See section on memory management.

Related Words: ALLOCBLOCK +STACK PUSH POP

-TRAILING (addr count -- addr count') "dash trailing"

-TRAILING adjusts the count of a string to remove trailing blanks. It does this by scanning backward from the last character toward the first character and the first non-blank character stops the scan.

```
" Hello     " COUNT .S -TRAILING .S TYPE
```

Related Words: COUNT TYPE SKIP

. (n --) "dot"

Print the integer number on the top of the stack. Use the value of BASE as the current numeric base. Print a space after the number.

```
2 3 + .  ( will print "5 " )
```

Related Words: U. .R D. D.R .HEX BASE ? N>TEXT F.

." (<text> --) "dot quote"

Print a text string terminated with a ".

If used inside a colon definition, the text will be compiled into the word for later printing. This word will also work outside of colon definitions although the Forth '83 standard says it should fail.

With buffered I/O, the output may not appear until you call CR KEY or FLUSHEMIT. See FAST .

```
." Hello world!"
```

Related Words: (.") TYPE CR

.. (struct-addr <member> -- member-addr) "dot dot"

Return the address of a member in a 'C' like structure. See chapter on the Amiga 'C' structure interface.

```
SCREEN MY-SCREEN ( declare screen structure )
MY-SCREEN .. SC_RASTPORT ( -- address-rastport-member)
```

File: JU:C_STRUCT

..! (n struct-addr <member> --) "dot dot store"

Set the member of a structure to n. C! , W! or ! will be used depending on the width of the member. See chapter on the Amiga 'C' structure interface.

```
NEWSCREEN MY-NEWSC ( declare screen structure )
3 MY-NEWSC ..! NS_DEPTH  ( set number of bitplanes )
```

File: JU:C_STRUCT

..@ (struct-addr <member> -- n) "dot dot fetch"

Fetch the value of a structure member. C@ , W@ or @ will be used depending on the width of the member. See chapter on the Amiga 'C' structure interface.

```
MY-WINDOW ..@ WD_WIDTH ( fetch width of window )
```

File: JU:C_STRUCT

.ELSE (--) "dot else"

Conditional compilation word used with .IF . See .IF

.ERR (--) "dot err"

.ERR is useful when reporting errors. It clears the stack, executes CR , then TYPEs the text at HERE, in preparation for your error message and subsequent QUIT.

```
: ?HaveError   ( flag -- , quits with error message if error )
    IF .ERR  ." flag was not Zero!"  QUIT
    THEN  ;

\ Here is an example with the resulting printout.
0 ?HaveError  ok
1 ?HaveError
?HAVEERROR ? ... flag was not Zero!
```

Related Words: HERE ABORT" QUIT TYPE

.FILE (<filename> --)

Uses DOLINES to print the contents of a file with line numbers. Similar to TYPEILE .

File: JU:DOLINES

.HEX (n --) "dot hex"

Print the top of stack as a hexadecimal, HEX , number regardless of the current base.

```
DECIMAL 17 .HEX  ( prints 11 )
```

.HX (n --) "dot h x"

Prints one digit. This will print a digit in any base up to 36.

Equivalent definition:

```
: .HX  ( n -- )
    DUP 9 > IF 37 + ELSE 30 + THEN EMIT ;
```

Related Words: .HEX .S

.IF (flag --) "dot if"

.IF , along with .ELSE and .THEN , allows for conditional compilation. These words work like IF ELSE THEN but they are immediate words and work in the interpret state as well. JForth source files include numerous examples of the use of these conditional words.

.IF and related operators are NOT nestable.

```
false .IF  : FOO ( one way ) ;
.ELSE  : FOO ( a different way ) ;
.THEN
```

Related Words: .IF .ELSE .NEED EXISTS? .THEN

.IMAGE (--) "dot image"

.IMAGE will report whether the current image is:

Relocatable (contains LONG-ABSOLUTE address references, and if so, how many)

OR

Position-Independent (the compiler has been able to use relative addressing modes for all threading).

A LONG-ABSOLUTE reference (JSR) will be compiled only if: 1) the calling program is over 32K away from the called program... AND 2) the called program is not resident in the bottom 96K of JForth.

Related Words: JSR-CODE MAP CFA,

.MODULES (--)

Display state of detachable modules. See chapter on modules.

.NEED (--) "dot need"

.NEED is a conditional compilation word that works with .THEN and .ELSE . If the word following .NEED is in the dictionary, .NEED will skip to the .ELSE or .THEN part of the conditional compilation structure. If the word does not exist, it will continue processing the text that follows.

```
.NEED WIERD ( WIERD ain't in the dictionary  ... )
    : WIERD   CR ." Like wow, man. " ;
.ELSE  cr ." this place is already weird"  ( WIERD is around)
```


! " # $ % & ` ( ) * + ' - . / 0-9 : ; < = > ? @ AZ [\ ] ^ _ az { | } ~

```
.THEN
```

Related Words: .IF .ELSE .THEN EXISTS?

.R (n width --) "dot r"

Print N right justified in a field WIDTH characters wide. This is useful for preparing formatted output like tables, etc. If the number is wider than WIDTH it will still print all the digits.

```
742  12  .R    ( prints  ___742 )
   ( 7 spaces to the left )
```

Related Words: D.R . # COMMAS

.RSTACK (--) "dot r stack"

Print the contents of the return stack.

File: JU:.RSTACK

.S (--) "dot s"

Print the contents of the data stack. The number on the right is the top of stack. The stack will be unchanged by the operation. Often placed inside troublesome words when debugging.

Related Words: DEPTH DEBUG SP@ .RSTACK

.THEN (--) "dot then"

Terminate a .IF or .NEED conditional construct. See .IF for more details.

/ (a b -- a/b) "slash" or "divide"

Divide the top two items on the stack and leave the result. This is an integer 32 bit divide. The answer will be truncated. Negative results will be rounded towards zero. Use /MOD if you want the remainder.

```
33 5 / . ( prints 6 )
-17 8 / . ( prints -2 )
```

Related Words: /MOD M/ CELL/ D2/ 2/ W/ U/ DU2/ U2/

/MOD (a b -- remainder a/b) "slash mod"

Performs 32 bit integer divide of A by B. Leave remainder and quotient.

```
73 10 /MOD .S  ( prints 3 7 )
```

Related Words: / MOD

/STRING (addr cnt index -- addr' cnt')"slash string"

/STRING indexes into the string at addr, returning the address and count of the remaining string.

```
" Hello World" COUNT 6 /STRING TYPE ( prints World )
```

---

0 (-- 0) "zero"

Defined as a constant to speed up compilation.

0" (<text> -- 0string) "zero quote"

Convert the following text to a NUL terminated string. The text should be terminated by a ". NUL, or zero, terminated strings are used by 'C' and are therefore needed when passing strings to the Amiga Libraries. The parameter 0string is the address of the first character in the string.

```
0" Hello" 0COUNT TYPE
```

Related Words: 0COUNT >ABS $>0 DOS0 " $APPEND 0FOPEN

0< (n -- flag) "zero less than"

Compares the top of stack to zero. Leave TRUE if less than zero, otherwise leaves FALSE.

```
: TEST 0< IF ." Negative!" THEN ;
-5 TEST ( print Negative!)
```

Related Words: 0> 0= +-

0= (n -- flag) "zero equals"

Compares the top of stack to zero. Leave TRUE if equal to zero, otherwise leaves FALSE. Can be used like NOT to complement a flag.

Related Words: 0> NOT IF

0> (n -- flag) "zero greater than"

Compares the top of stack to zero. Leave TRUE if greater than zero, otherwise leaves FALSE.

Related Words: 0< 0=

0BRANCH (flag --) "zero branch"

0BRANCH is compiled into a definition by IF UNTIL WHILE to do a conditional branch. If flag = FALSE, the program will branch to another location. If flag = TRUE, execution continues with the next instruction.

Standards: FIG, called ?BRANCH in '83 and '79 .

Related Words: IF WHILE UNTIL BRANCH NOT0BRANCH

0COUNT (0string -- addr count) "zero count"

Count the characters in a NUL terminated string.

```
0" Hello" 0COUNT TYPE
```

0FOPEN (0name -- filepointer | 0) "zero f open"

This word takes the address of a NULL-TERMINATED string and submits it to AmigaDOS to request it be opened as a filename. If the file can be successfully opened, its pointer is returned, else a FALSE is returned. This pointer may be passed in other file utilities to initiate reads, writes, seeks, etc.

Please see the chapter on File I/O for more information.

Related Words: FILE? FOPEN $FOPEN FREAD

0RP (--) "zero r p"

0RP initializes RP to the value in the user variable RP0 . RP = return stack pointer. This is usually used in error recovery.

Warning: This clears the subroutine calling stack of the 680x0 so be very careful when using this.

Related Words: RP0 RP!

0SP (a? b? whatever -- , empty) "zero s p"

Clear the data stack by initializing SP to the value in the user variable SP0 . SP = stack pointer. Remember the top of stack is cached in D7. Only recommended for interactive use and in error recovery.

```
11 22 33 0SP  ( stack now empty )
```

Related Words: SP0 SP!

---

1 (-- 1)

Defined as a constant to speed compilation.

1+ (n -- n+1) "one plus"

Add one to the value n on top of the stack.

```
7 1+ . ( print 8 )
```

Related Words: 2+ 1- 2- 2* 2/ + -

1- (n -- n-1) "one minus"

Subtract one from the value n on top of the stack.

Related Words: + - 1+ 2+ 2- 1 2

---

2 (-- 2)

Defined as a constant to speed compilation.

2! (d addr --) "two store"

Store a 64 bit double number at the given address. Only even addresses are allowed on a 68000 system. Use ODDD! for odd address double stores. 2! is a synonym for D! .

Related Words: D! ODD-D!

2* (n -- n*2) "two times"

Multiply top of stack by 2.

```
-5 2* . ( print -10 ) 5 2* . ( print 10 )
```

2N (n -- 2**n) "two to the n"**

Raise two to the Nth power.

```
3 2**N . ( print '8'  = 2*2*2 )
```

File: JU:BSORT

2+ (n -- n+2) "two plus"

Add two to the value n on the top of the stack.

```
85  2+  .  ( prints 87 )
```

2- (n -- n-2) "two minus"

Subtract two from the value n on the top of the stack.

2/ (n -- n/2) "two slash"

Divide the top of stack by two. This uses an arithmetic left shift which is much faster than the normal divide.

Related Words: 2* U2* U2/ DU2/ DU2* W/ /MOD U/ M* U* D2/ / M/ DIGS/ */MOD

2@ (addr -- d) "two fetch"

Fetch a double number from the given address. Only even addresses are allowed. See D@ .

Related Words: D@ ODD-D@ X@

2DROP (a b --)

Drop 2 cells from top of stack.

Standards: '83 and FIG . Called DDROP in '79 .

Related Words: DROP

2DUP (a b -- a b a b)

Duplicates the top pair of stack values.

Standards: '83 and FIG . Called DDUP in '79 .

Related Words: DUP

2OVER (a b c d -- a b c d a b)

Copies pair of numbers over top pair.

Related Words: OVER DUP 2DUP DOVER

2SORT (n1 n2 -- smallest biggest) "2 sort"

Sort two numbers.

Related Words: -2SORT MIN MAX

2SWAP (a b c d -- c d a b)

Swaps the top two pairs of stack values.

Related Words: SWAP 2DUP

---

3 (-- 3)

Defined as a constant to speed compilation.

32K (-- 32768) "thirty-two k"

Constant equal to 32 * 1024.

64K (-- 65,536) "sixty four k"

Constant equal to 64 * 1024

: (<name> --) "colon "

tart the definition of a new Forth word. Enters COMPILE mode by setting STATE to TRUE. The code that follows will be compiled into the word. Sets CONTEXT and CURRENT vocabularies to the same. Terminate with ; which sets SMUDGE bit .

```
\ Define a new Forth function then test it.
: DOUBLE ( n -- n*2 )
    DUP + ;
7 DOUBLE .  ( prints 14 )
```

Related Words: ; CREATE

:CREATE (<name> --) "colon create"

:CREATE acts to create a dictionary header, using the next word available in the input stream. It is used by: CREATE VARIABLE CONSTANT and other defining words. :CREATE is a DEFERred word. See (CREATE).

Related Words: (CREATE)

:LIBRARY (<name> --) "colon library"

Define a new Amiga Library to be called using CALL . Any Library that has an FD file can be called from JForth. The standard libraries have already been defined for JForth.

```
:LIBRARY STUFF  ( define new library )
STUFF?  ( open library )
```

See the chapter on "Calling Amiga Library Routines".

:STRUCT (<name> --)

Defines a new structure. See the special section on interfacing with the Amiga.

File: JU:C_STRUCT.

Related Words: ;STRUCT .. ..@ ..!

; (--) "semicolon"

Terminate a colon definition. Compiles EXIT into the definition being created. EXIT is the execution time procedure of ; Clears the smudge bit. Set STATE back to zero. Set size field to reflect size of word. Places the system back into interpret mode. ; is immediate.

```
: HI ." Hello!" CR ;
```

Related Words: EXIT : BOTH INLINE :CREATE

;STRUCT (--) "semicolon struct"

Terminate a 'C' like structure definition. Save a pointer to the last member defined for DST to use. See chapter on "Accessing Amiga 'C' Structures".

File: JU:C_STRUCT

< (n1 n2 -- flag) "less than"

Compare n2 with n1. Leaves TRUE if n1<n2 , FALSE if not.

```
7 23 <  ( leaves TRUE )
```

Related Words: > = U< <=

<# (d --) "less sharp"

<# starts numeric output string conversion. Initializes the pointer in HLD to point to the beginning of PAD. Terminate with a #> . See # .

```
S->D <# # # # # #S #>  ( gives at least 5 digits)
```

Related Words: #S # #> SIGN HOLD HLD

<= (n1 n2 -- flag) "less than or equal"

Compare n2 with n1. Leaves TRUE if n1 <= n2 , FALSE if not.

```
7 23 <=  ( leaves TRUE )
23 23 <= ( leaves TRUE )
```

Related Words: > = U< < >=

<FASTEMIT> (char --) "bracket fast emit"

Transmits one character to the output device using buffered I/O. This is the default contents of the DEFERred word EMIT.

<FASTEMIT> is installed into EMIT when FAST I/O mode is entered. To speed up printing, the character will be held in a buffer until:

```
the FAST I/O buffer is filled,
a CR is executed,
or KEY , EXPECT or FLUSHEMIT is executed.
```

This is faster than outputting characters one at a time. <FASTEMIT> calls +OUT to adjust OUT to reflect the current cursor position. This is done on each character.

Related Words: EMIT FAST SLOW FLUSHEMIT

<FLUSHEMIT> (--) "bracket flush emit"

This is the default contents of the DEFERred word, FLUSHEMIT.

When executed in FAST mode, <FLUSHEMIT> sends any characters held in the FAST buffer to be sent to the standard EMIT device.

= (n1 n2 -- flag) "equals"

Compares top two values n1 and n2 on stack and replaces them with flag = true if they are equal and flag = false if they are not equal.

Related Words: < > 0= -

> (n1 n2 -- flag) "greater than"

Compare top two values on stack. Leave TRUE if n1 greater than n2, otherwise leave FALSE.

```
23 7 > ( leave TRUE )
```

Related Words: < = U> IF >=

>= (n1 n2 -- flag) "greater than or equal"

Compare top two values on stack. Leave TRUE if n1 >= n2, otherwise leave FALSE.

```
23 7 >= ( leave TRUE )
23 23 >= ( leaves TRUE )
```

Related Words: < = U> IF >

>ABS (rel-addr -- abs-addr) "to absolute"

Convert JForth relative address to 68000 absolute address.

JForth "addresses" are actually an offset from the base of the dictionary. This allows us to use addresses that stay the same every time we run JForth, even though JForth may load into different places in memory each time. This makes JForth code more relocatable. The Amiga operating system, however, uses normal 68000 addresses. You must convert JForth relative addresses to 68000 absolute addresses before passing them to the Amiga.

```
DATE-VARIABLES @ >ABS ( Amiga addr )
CALL DOS_LIB DATESTAMP
```

Related Words: >REL S@

>BODY (cfa -- body) "to body"

Convert the CFA or tick address (start of the executable code) of a CREATE or CREATE/DOES> child to its Body address (location of the related data). See BODY>.

>BODY will only return a meaningful result if the passed-in address points to the CFA of a word created via CREATE, either directly or via a CREATE/DOES> defining word. This does NOT include the CFA for other data-structures, such as VARIABLEs, CONSTANTs, or VALUEs.

[Note: In JForth Version 1.2 and earlier, **>BODY** was a noop, implying the **body** and the **cfa** were the same!]

Related Words: ' BODY> >NAME NAME> DO-DOES-SIZE

>DOS (addr count --) "to dos"

Convert the string residing at ADDR that is COUNT bytes long to a NUL-TERMINATED string in the DOS0 buffer. This can then be passed to a DOS Library routine.

```
( moves a filename to the DOS0 buffer )
" df0:c/dir" COUNT >DOS
DOS0 0COUNT TYPE
```

Related Words: DOS0 +DOS $>0

>IN (-- addr) "to in"

This is the user-variable that stores the index that INTERPRET uses to track its current place in the TIB.

>IN is set to 0 by QUERY, and incremented as INTERPRET processes the input stream.

```
: TYPE&EXEC  ( <name> -- , type name then execute. )
    >IN @  ( save location )
    CR 32 WORD COUNT TYPE CR
    >IN !  ( restore location for interpreter to execute )
;
: FOO ." Hello" CR ;
TYPE&EXEC FOO
```

Related Words: TIB EXPECT BLK PULLTIB PUSHTIB

>INLINE to inline

```
( relative-altered-return-address -- )
( --R-- absolute-return-address )
```

>INLINE takes a relative address and converts it to an absolute address before putting it back on the return stack.

>INLINE is part of a set of 4 words that allow for transportable inline data words. >INLINE puts an altered inline data address back to the top of the return stack, in the proper return stack format. In JForth for the Amiga, the return stack contains absolute addresses, but @ and other data movement words are relative. See chapter on transportability.

Related Words: INLINE> INLINE@ INLINE+

>LINK (cfa -- link) "to link"

>LINK converts a CFA to a LINK address.

Related Words: >NAME N>LINK

>NAME (cfa -- nfa) "to name'

>NAME converts the code-field-address of a dictionary header (that returned by ') to the name-field-address.

```
' SWAP  >NAME ID.
```

Related Words: ' NAME>

>NEWLINE (--) "to new line'

Perform a CR unless the cursor is already at the start of a new line.

Related Words: LINELIMIT OUT CR

>PARENT (child-CFA -- parent-CFA) "to parent"

>PARENT is a JForth unique word to fill the need for a transportable way to convert a child CFA to its parent CFA . See related words.

Related Words: CREATE DOES>

>R (n --) (--R-- n) "to r"

>R pops a value from top of stack and pushes the value on top of return stack. You can use the return stack as a place to store temporary variables. You must clean up before the end of a definition, however, or your program will crash.

```
: EXAMPLE ( n m -- n+1 m )
    >R 1+ R> ;    ( faster than SWAP 1+ SWAP )
```

Related Words: R> X>R XR> XRDROP R@ RDROP RPICK

>REL (absolute-addr -- relative-addr) "to rel"

Inverse of >ABS . Converts an absolute 68000 address to a relative JForth address. Addresses returned by Amiga Library routines must be converted to relative before accessing them with JForth.

Related Words: >ABS

>US (n --) (---user-stack--- n) "to user"

Push N onto the User Stack. The User Stack is another stack that can be used as a convenient place to push data. It is used by during compilation by IF DO and other conditional words.

Related Words: R> >R US> US@

? (addr --) "question"

Print the value stored at ADDR . Defined as : ? @ . ;

```
VARIABLE VAR1   789 VAR1 !
VAR1 ?
```

?ABORT" (flag --) "question abort quote"

?ABORT is a JForth synonym for ABORT" .

?CLOSEBOX (-- flag) "question close box"

Return TRUE if the Close Box gadget has been hit in an open graphics window. The window must have been made current using GR.SET.CURWINDOW. See chapter on Graphics and Event handling, also JD:DEMO_BOXES.

File: JU:AMIGA_EVENTS

?CONTROL-D (-- flag) "question control d"

Return TRUE if control D has been hit on the keyboard. Used in the debugger to interrupt execution and enter debug mode.

File: JU:DEBUGGER

?COMP (--) "question comp"

?COMP issues an error message and QUITs if JForth is not in compile mode, ie. STATE is TRUE. Use to ensure that the system is in compile mode.

Related Words: STATE ?EXEC

?CSP (--) "question c s p"

?CSP issues an error and QUITs if the current stack pointer position does not equal the value stored in CSP. Uses WARNING and MESSAGE. Used for compiler security because an unbalanced stack means compilation error. JForth uses !CSP to save the stack pointer position upon start of compilation and ?CSP to check it when finished compiling.

Related Words: DEPTH SP@

?DUP (n -- n n | 0) "question dup"

Duplicates top of stack if non-zero. Same as -DUP .

Related Words: DUP 0=

?ERROR (flag error-number --) "question error"

?ERROR issues error message specified by N and QUITs if flag = true.

Related Words: ERROR MESSAGE WARNING QUIT ABORT (ABORT)

?EXEC (--) "question execute"

?EXEC prints error message and QUITs if system is not in execute mode. STATE = zero if system is in execute mode. : uses ?EXEC

Related Words: ?COMP STATE

?EXIT (flag --) "question exit"

?EXIT will exit a word if flag on stack is true, else it will continue. RETURN is safer, since it will compile ?EXIT if not within a loop, and it will compile <?RETURN> if within any level of nested do loops.

Related Words: EXIT RETURN ?RETURN

?GOTO.ERROR (flag --)

Jump to ERROR: label if flag is true.

File: JU:GOTO_ERROR

Related Words: GOTO.ERROR

?LEAVE (flag --)

LEAVE a DO LOOP if the flag is true. See LEAVE .

Related Words: LEAVE DO LOOP RETURN

?LETTER (char -- flag)

If the ASCII character on the stack is alphabetic, return a TRUE; otherwise return FALSE.

Related Words: NUMBER? ASCII

?OF (value flag -- value |)

Executes the following code up to an ENDOF, and drops the value if the flag is true. See CASE.

?PAIRS (n1 n2 --) "question pairs"

Tests n2 and n1 for equality. If equal, they are discarded and execution goes on. If not equal, an error message is given and QUIT is executed. It is used for checking whether conditional constructs are paired correctly. Usual message is CONDITIONALS NOT PAIRED.

Related Words: ?DO_FLAG ?IF_FLAG AGAIN THEN ELSE LOOP

?PAUSE (--) "question pause"

Pause if a key has been hit. ?PAUSE calls ?terminal . If a key has been activated, it will stop and wait for a line of text followed by a carriage return. It will execute the statements in that line then continue after where the ?PAUSE was executed. It is used by VLIST DUMP TYPEFILE and some other printing words.

Related Words: ?TERMINAL KEY

?RETURN (flag --) "question return"

?RETURN takes a flag and returns if the flag is non-zero. See RETURN.

Related Words: RETURN EXIT ?EXIT EXIT

?STACK (--) "question stack"

?STACK executes ?ERROR if stack underflow or overflow occurs.

An UNDERFLOW occurs if more items are removed from the stack than are actually on the stack. An OVERFLOW occurs if so many items are added to the stack that it grows down into the PAD memory area.

?STAY (stay-flag --) "question stay"

?STAY is the opposite logic version of ?LEAVE; ?STAY leaves if the flag is false.

?TERMINAL (-- key-hit?) "question terminal"

Check the keyboard to see if a key has been hit. Return TRUE if so. KEY can then be used to read the character. ?TERMINAL is deferred.

```
: YAK BEGIN ." Yak Yak Yak" CR ?TERMINAL UNTIL ;
```

Related Words: ?PAUSE ?CLOSEBOX ?CONTROL-D

?VISIBLE (char -- flag) "question visible"

If the character on the stack is a visible, printable character (would cause the cursor position to move to the right one column) return TRUE; otherwise return FALSE.

```
: SAFE.EMIT ( char -- , emit if visible, '.' if not )
    DUP ?VISIBLE NOT
    IF DROP ASCII . THEN EMIT ;
```

@ (addr -- n) "fetch"

Fetch the 32 bit value N from the address on the stack. This is the primary operator for reading memory. The address is a relative JForth address. The address must be even.

```
VARIABLE VAR1   567 VAR1 !
VAR1 @ .   ( print 567 )
```

Related Words: C@ W@ ! >REL ODD@

! " # $ % & ` ( ) * + ' - . / 0-9 : ; < = > ? @ AZ [\ ] ^ _ az { | } ~

@&CLOSEFILES (var-addr --) "fetch and close files"

This word accepts the address of a VARIABLE or USER-variable that contains the address of a memory block containing a list of file pointers.

An application can easily maintain a list of its open files by calling ALLOCBLOCK and using the area as a stack, pushing each file-pointer as it receives them. All of the files may be closed at once by passing the VARIABLE or USER-variable pointing to this area to @&CLOSEFILES.

If the address passed to @&CLOSEFILES contains zero, there will be no effect.

Once @&CLOSEFILES has closed the files, it will free the memory block used to store the list, and store a zero in the pointer addr.

Related Words: @ ALLOCBLOCK FCLOSE

@&FREEBLOCKS (var-addr --) "fetch and free blocks"

This word accepts the address of a VARIABLE or USER-variable that contains the address of a memory block containing a list of allocated memory blocks.

An application can easily maintain a list of its allocated memory blocks by calling ALLOCBLOCK and using the area as a stack, pushing each block pointer as it receives them. All of the blocks may later be freed at once by passing the VARIABLE or USER-variable pointing to this area to @&FREEBLOCKS.

If the address passed to @&FREEBLOCKS contains zero, there will be no effect.

Once @&FREEBLOCKS has freed the areas in the list, it will free the memory block used to store the list, and store a zero in the pointer addr.

Related Words: ALLOCBLOCK FREEBLOCK

@EXECUTE (addr --) "fetch execute"

Fetch a CFA from the given address and execute that word. See EXECUTE.

ABORT (--)

Abort execution, return to outer interpreter. No message is displayed.

```
: EXAMPLE ( N -- )  1,000,000 >
   IF  CR ." Oh my God! We're going TOO HIGH!!!" CR
       ABORT  ( ABORT" would be better here )
       ( will QUIT with no "OK" message )
   THEN ;
```

ABORT is a deferred word that normally calls QUIT.

Related Words: ABORT" WARNING" .ERR QUIT

ABORT" (flag <message> --) "abort quote"

Display message and abort if the flag is true.

```
: EXAMPLE ( N -- )  1,000,000 >
   ABORT" Oh my God, We're going TOO HIGH!!!" ;
```

Related Words: ABORT QUIT ?ABORT" WARNING"

ABS (n -- |n|) "absolute"

Calculate absolute value of N.

```
5 ABS .
-5 ABS .  ( both will print positive 5 )
```

Related Words: DABS

ABS! (n absolute-address --) "a b s store"

Store longword, word, or byte at an absolute 68000 absolute address. Equivalent to >REL ! but faster. Useful for accessing hardware.

Related Words: >ABS ! W! C! ABS@ >REL

ABSW! (n absolute-address --) "a b s w store"

See ABS!

ABSC! (n absolute-address --) "a b s c store"

See ABS!

ABS@ (absolute-address -- n) "a b s fetch"

ABSW@ (absolute-address -- n) "a b s w fetch"

ABSC@ (absolute-address -- n) "a b s c fetch"

Fetches longword, word, or byte from absolute address. Equivalent to >REL @ but faster. Useful for accessing hardware.

Related Words: >ABS @ W@ C@ ABS!

ADST (absolute-addr <structure> --)

Dump a structure whose absolute address is on the stack. Similar to DST which takes a relative address. See DST .

File: JU:DUMP_STRUCT

AGAIN (--)

AGAIN at execution time: marks the end of an infinite loop and branches back to its corresponding BEGIN . Warning - an infinite loop won't stop until QUIT or RETURN is executed.

At compile time : compiles a BRANCH into the dictionary. Resolves the loop entry point address provided by BEGIN into a return branch offset and stores this offset in the dictionary. AGAIN and BEGIN are paired. Error is detected by ?PAIRS if no match.

```
\ Warning this word will never stop!!!
: ETERNITY BEGIN ." Forever is a long time!" AGAIN ;
```

Related Words: BEGIN BACK BRANCH DO UNTIL WHILE

ALIAS (<oldname> <newname> --)

Create an alternative name for an existing word.

```
ALIAS 2DUP DDUP
```

ALIGN (--)

Word-align the dictionary pointer DP .

ALIGN is used at the end of any dictionary-allocation process that may have left HERE at an odd address (the 68000 CPU and JForth both require the dictionary to be word aligned). ALIGN will allocate a byte in the dictionary if DP is odd; no effect if even.

Related Words: C, CARRAY ALLOT EVEN-UP

ALLOCBLOCK (memtype size -- addr | false)

Allocates a memory area of the given size and type and returns the JForth-relative address of the block. If the call for memory is not successful, a FALSE flag is returned.

The programmer will call ALLOCBLOCK to get an area of memory from the free memory list maintained by AmigaDOS. JForth will remember the block and return it to AmigaDOS at BYE , if the programmer does not.

This call is a direct interface to the Amiga EXEC function, AllocMem. It returns, however, a JForth relative address, usable by all the JForth memory-reference words, such as @, C@, !, C!, etc.

The type parameter is referenced as in Amiga literature:

```
MEMF_CHIP   - memory in lowest 512K, accessible by graphics
              and sound hardware.
MEMF_FAST   - memory above 512K, cannot be used by hardware
              chips.
MEMF_PUBLIC - memory is to be used for different tasks or
              interrupt code, and may apply to task control
              blocks, messages, ports, etc.
```

```
MEMF_CLEAR  - this parameter may be specified to clear the
              memory upon allocation; otherwise no
              initialization is done.
```

These parameters may be combined (as long as they are not self-cancelling) by logically ORing them together. For example, to indicate a global area that may be used by the graphics chips, this argument would be formed:

```
MEMF_CHIP    MEMF_PUBLIC  OR
```

ALLOCBLOCK also provides a range of operators that may be used to expand the functionality of the memory block to act as stacks and buffers, and/or automatically be freed at QUIT .

The address returned from a successful call may subsequently be passed to FREEBLOCK to return the memory to AmigaDOS when no longer needed.

Later versions of the Amiga may support 1024K of CHIP RAM.

```
Related Words: FREEBLOCK  MARKFREEBLOCK  MEMF_CHIP  MEMF_FAST
MEMF_PUBLIC MEMF_CLEAR FREEBYTE FREEBYTEA PUSH POP -STACK +STACK
SIZEMEM
```

ALLOCBLOCK? (memtype size -- addr)

Identical to ALLOCBLOCK except this word reports an error and aborts if memory could not be allocated.

ALLOT (numbytes --)

ALLOT allocates space in the dictionary by advancing the dictionary pointer DP by N bytes.

NOTE: if you ALLOT an odd number of bytes, the word ALIGN must be called before any operations involving HERE are allowed to occur. It does not hurt to call ALIGN for even numbers so use ALIGN liberally.

VARIABLE CHEAP-ARRAY 96 ALLOT (has room for 100 bytes)

```
Related Words: DP  HERE  ALIGN
```

ALSO (voc --voc-stack-- voc voc)

Duplicate top of vocabulary stack. Used to make room for vocabulary in the current context. See the section on vocabularies.

```
VOCABULARY MUSIC  ( create a new vocabulary )
ALSO MUSIC  ORDER ( also search the MUSIC vocabulary )
PREVIOUS     ( put things back the way they were )
```

AND (a b -- a&b)

Does logical AND on each pair of corresponding bits in A and B. Both bits an A AND B must be 1 for bit in result to be 1. Useful for masking data.

```
HEX 7E53 FF AND .  ( print 53 )
: EVEN?  ( N -- flag , true if odd ) 1 AND 0= ;
```

```
Related Words: OR XOR
```

ANEW (<name> --) "a new"

Forgets a word if already defined, no effect otherwise. This is often used at the beginning of a file that is under development. Every time you recompile the file, it will automatically forget the code compiled the last time. The filename is typically appended to the prefix TASK- .

```
\ Roll dice.
INCLUDE? CHOOSE JU:RANDOM
ANEW TASK-DICE
: ROLL.DICE ( -- N )
    6 CHOOSE 1+
;
```

Load the file by entering:

```
INCLUDE DICE
ROLL.DICE .
```

You can now edit this file, adding and changing code. You can then include this file again and again without getting multiple definitions of your code. Notice that the INCLUDE? is placed before the ANEW. This is so CHOOSE won't be forgotten each time.

One thing to remember in using ANEW is that if you load a bunch of files after loading DICE , then reload DICE, you will lose the definitions from the other files. (The actual files will be unaffected.) This is only a problem when you are loading a file that stops because it contains a word that is undefined. If you load the file that contains that undefined word, and then try to reload your file, the first thing ANEW will do is forget the code you just loaded. The same word will show up missing agin. You can while away many a rainy day stuck in this loop. The thing to do is to first FORGET your task word. Consider the following sequence:

```
INCLUDE MYFILE
( assume myfile crashes because of an undefined word )
( edit myfile to do an INCLUDE? for that word)
FORGET TASK-MYFILE
INCLUDE MYFILE
```

Unless you have more undefined words, you can continue just using INCLUDE MYFILE .

Related Words: FORGET IF.FORGOTTEN

ANSI.BACKWARDS (n --)

Move cursor N characters backwards. This is only one of a number of ANSI based "terminal editing" commands that can be found in the file JU:ANSI.

APTR (<name> --) "a pointer"

Define a 32 bit pointer member in a structure. See the section on Amiga 'C' structures.

File: JU:MEMBER

AREGS>ABS (--) " a regs to a b s"

Set the flag CONVERT-AREGS so that the very next Amiga CALL will convert parameters in address registers to absolute before passing them to the Amiga. This will save you having to call >ABS excessively. See chapter on Calling Amiga Libraries.

ARGS (<library_lib> <routine_name> --)

ARGS looks up arguments for an Amiga Library call and displays them. It provides a form of automatic documentation. It looks in the FD files so there must be one for the library you specify. The library need not be open.

```
ARGS GRAPHICS_LIB DRAW

Will display:  Draw(rastPort,x,y)(A1,D0/D1)
```

This tells you that the graphics DRAW routine takes 3 arguments. Place the absolute address of a RASTPORT on the stack followed by x and y values, then use CALL to invoke the routine. JForth will automatically build the necessary code to stuff the 68000 registers and make the call.

Related Words: CALL DCALL

ARRAY (numcells <name> --)

Creates a one dimensional array of length NUMCELLS of 32 bit values starting at the next available dictionary location . Cells are initialized at compile time to zero's. The index of the first array item is zero just like in 'C'. Let's create an array with 20 items.

```
20 ARRAY MY-ARRAY
```

The array that was created has the following stack diagram:

MY-ARRAY (index -- addr)

Now lets store a value in that array and then fetch it back.

```
731 12 MY-ARRAY ! ( store 731 into cell 12 of MY-ARRAY )
0 MY-ARRAY @  ( get first cell value )
```

(ODE, the object oriented dialect, has a fancier type of array.)

Related Words: ALLOT ALLOCBLOCK OB.ARRAY ARRAYOF

ARRAYOF (n <structure> <name> --)

Create an array of structures. For example:

```
10 ARRAYOF GADGET MY-GADGETS
```

ASCII (<char> -- ASCII-value)

Converts the next character in the input stream to its ASCII equivalent and puts the value on top of the stack .

```
ASCII A .  ( print 65 )
```

ASHIFT (n shift-count -- n-shifted)

Arithmetic shift N by SHIFT-COUNT. Shift left is shift-count is positive, right if negative. Preserve the sign of N when shifting left by "dragging" the sign bit.

```
-20 -2 ASHIFT . ( print -5 )
```

The word SHIFT is similar but does not preserve sign.

ASM (-- , <wordname>)

See the chapter on 68000 ASSEMBLY.

AUTO.INIT (--)

Used for automatic initialization. You write this word. When JForth starts up, it searches the dictionary for a word called AUTO.INIT and executes the first one it finds. If you have an initialization word that you would like called at startup, define a word called AUTO.INIT that calls the previous AUTO.INIT then calls your word. This will add onto a chain of AUTO.INITS that initialize many parts of JForth.

Code that allocates memory, opens files or libraries, initializes hardware, opens windows or builds tables is often called with AUTO.INIT . The matching terminate code is then called from AUTO.TERM. In general, DO NOT allocate memory open files or windows, etc. at compile time. Place this code in a colon definition and use INIT words.

Here is an example that shows how to automatically initialize a system, as well as automatically clean up on BYE or if the code is forgotten.

```
( variable to avoid double init or term )
VARIABLE IF-MY-INIT
: MY.STARTUP  IF-MY-INIT @ 0=
    IF  ALLOC.STUFF SET.STUFF
        IF-MY-INIT ON
    THEN
;
: AUTO.INIT ( -- , add my.startup to init chain )
    AUTO.INIT ( important!!)
    MY.STARTUP
;
: MY.TERM  IF-MY-INIT @
    IF  CLEANUP.STUFF FREE.STUFF
        IF-MY-INIT OFF
    THEN
;
: AUTO.TERM MY.TERM AUTO.TERM ; ( called on BYE )
IF.FORGOTTEN MY.TERM  ( call MY.TERM if forgotten )
```

Related Words: AUTO.TERM IF.FORGOTTEN

AUTO.REQUEST ($body $posi $nega -- flag)

Put an Amiga Auto Requester on the screen. The body will be the main message. There will be two possible responses, one positive and one negative. These might be "Yes" and "No", or "OK" and "Cancel". When the user selects a response, a TRUE will be returned for a positive response, otherwise FALSE.

File: JU:AUTO_REQUEST

AUTO.TERM (--)

This word is called automatically when JForth exits using BYE. You can define a word called AUTO.TERM to cleanup your code on BYE. See AUTO.INIT for an example of it's use.

B->S (byte -- sign-extended-byte-value) "b to s"

Sign extend a byte value to 32 bits.

```
HEX E7 B->S .HEX ( prints FFFFFFE7 )
```

Related Words: W->S S->D

BACK (addr --)

BACK resolves the supplied address ADDR into a backward branch offset relative to HERE and compiles the offset into the dictionary. It is used internally to compile UNTIL , AGAIN , etc.

BASE (-- addr)

BASE is a user variable that contains the current number base used for input and output number conversion. HEX stores sixteen into BASE. DECIMAL stores ten into BASE.

The JForth ok prompt will tell you about the current base:

```
ok      - decimal
ok(hex) - hexadecimal
ok(bin) - binary
ok(7)   - in base 7
```

To display the base in decimal without destroying contents of BASE:

```
: BASE? BASE @ DUP DECIMAL . BASE ! ;

DECIMAL 7 BASE !
6 1 + .  ( print 10 )
```

BEGIN (--)

BEGIN marks the beginning of a loop. May be executed indefinitely as opposed to a DO loop which is limited to a certain number of passes. Used with UNTIL , WHILE , REPEAT and AGAIN.

At compile time, BEGIN places the address of the next available dictionary location onto the stack so UNTIL or REPEAT or AGAIN can compile a return branch into their definition. Begin_flag is checked by UNTIL or REPEAT or AGAIN for a balanced loop.

compile time (-- entry_point_address begin_flag)

```
entry_point_address = location of first word of loop.
begin_flag is for compiler security.

BEGIN code ( -- flag ) UNTIL ( quits when true )
BEGIN code  AGAIN
BEGIN code ( -- flag ) WHILE code-if-true REPEAT

: JABBER BEGIN ." Blah blah! " ?TERMINAL UNTIL ;
```

BEGIN_FLAG (-- n)

A constant left by the compile time part of BEGIN and used by the compile time part of UNTIL, REPEAT, WHILE , or AGAIN .

BENCH (<forthword> --)

Benchmark a word, subtracting the overhead time found using BENCH.WITH. For example, many words are benchmarked by calling them many times from a DO LOOP. To get an accurate measurement of the word we need to subtract the time for an empty DO LOOP.

```
\ Measure speed of swap.
1,000,000 CONSTANT 1MEG
: TDO   1MEG 0 DO LOOP ; ( empty loop )
: TSWAP 1MEG 0 DO SWAP LOOP ;
BENCH.WITH TDO
23 45 BENCH TSWAP
```

Related Words: MEASURE BENCH.WITH '

BENCH.WITH (<forthword> --)

Determine overhead time for benchmarking Forth words. See BENCH for example.

BINARY (--)

Sets user variable BASE to 2 for binary input and output numeric conversion. The Forth ok prompt will be followed by a (bin) when BASE is set to binary.

Related Words: HEX DECIMAL BASE

BIT-SET? (n bit# -- flag)

Test to see if a bit in N is equal to 1. The rightmost, least significant bit is bit number zero.

```
HEX 84 2 BIT-SET? ( true )
```

Related Words: SET-BIT AND OR XOR

BL (-- 32) "b l" or "blank"

Constant equal to the ASCII value for a blank, or space.

```
: SAYAGAIN BL WORD COUNT TYPE ;
SAYAGAIN hello  ( prints HELLO )
```

Related Words: WORD -FIND

BLK (-- addr) "b l k"

A user-variable, used only in the BLOCK environment to hold which 1024 byte section of a SCREEN-FILE is being loaded and interpreted . BLK = 0 if not interpreting from a screen file.

WORD looks at BLK to determine the address of input data. QUIT sets BLK to 0.

Related Words: LOAD-FILE >IN BLOCK

BLKERR (--) "block error"

This word is the default contents for the DEFERed word, BLOCK.

It serves to print the text at HERE, followed by the message " ... BLOCK not initialized!", finally executing QUIT.

This sequence warns that an attempt to execute BLOCK has been made without having loaded the "JU:BLOCK" file. This file defines the support environment for BLOCK, defining the correct handler to replace BLKERR in the vector of BLOCK.

Related Words: BLOCK QUIT HERE

BLOCK (block# -- buffer-addr)

Returns the address at the start of the 1024 byte buffer numbered BLOCK# . If this block is not already loaded, it loads it from the file providing a system similar to virtual memory. Since JForth normally uses regular ASCII text files, you must load JU:BLOCK to use this facility.

Related Words: BUFFERADR BLK R# LOAD (FIRST) FLUSH INCLUDE

BODY> (data-addr -- CFA) "body from"

Convert the Body address (location of the data area) of a CREATE or CREATE/DOES> child to its CFA or tick address (start of the executable code). See >BODY.

BODY> will only return a meaningful result if the passed-in address points to the data field of a word created via CREATE, either directly or via a CREATE/DOES> defining word. This does NOT include the data location for other data-structures, such as VARIABLEs, CONSTANTs, or VALUEs.

[Note: In JForth Version 1.2 and earlier, **BODY>** was a noop, implying the **body** and the **cfa** were the same!]

Related Words: >BODY ' >NAME >LINK

BOTH (--)

BOTH is an immediate compiler directive used just preceding a ; at the end of a definition. Once stated, the compiler will verify that the word being compiled may safely be used as an inline definition.

If so, the word is initialized such that the compiler will either compile an indirect call to it, or move it inline, depending on the value of the user-variable MAX-INLINE when it is being compiled.

If not, a warning message is issued, informing the operator that the word cannot be compiled inline, and the word will be set as CALLED .

```
: EXAMPLE  ( -- )  ROT OVER +    BOTH ;
```

Related Words: ; INLINE MAX-INLINE CFA,

BRANCH (--)

Used internally. BRANCH is compiled by certain conditional words to re-direct program flow at run time. BRANCH adds a + or - offset to IP causing an unconditional jump.

Related Words: ELSE AGAIN REPEAT

BSIN (-- addr) "b s in" or "back space in"

Variable containing the value of the backspace character for input. Usually 08 hex is the default value.

Useful for working with different terminals.

Related Words: BSOUT

BSORT (#items --) "b sort"

Sort items using Batcher sort. To control what gets sorted, you must set the deferred word BSORT-EXCH? . For more information, see the file JU:BSORT and the definition of BSORT-EXCH?

File: JU:BSORT, see also JA:SORTMERGE

Related Words: BSORT-EXCH? 2SORT

BSORT-EXCH? (index-a index-b --) "b sort dash exchange question"

Called from BSORT. To use BSORT you must write a word that takes the index of two items, and exchanges them if they are in the wrong order. Then set BSORT-EXCH? to call your word. The items that are sorted can be anything including strings, integers, etc.

You could, for example, have a table that contained pointers to entries in a mailing address data base. To sort your table by zip code, write a word that accepts two indices into that table, compares the two mailing addresses they point to, then swap the table entries if they are out of order. Then set BSORT-EXCH? to call your word. When you then call BSORT, it will generate a pattern of indices that will eventually result in your entire table being sorted in a very short time. Please see the demo file JD:DEMO_GSORT for another example.

```
: MY-EXCH?   ( ia ib -- )
    2DUP OUT-OF-ORDER?  ( you must supply this word )
    IF EXCHANGE-THEM    ( you must supply this word )
    ELSE 2DROP
    THEN
;
' MY-EXCH? IS BSORT-EXCH?
25 BSORT
```

File: JU:BSORT

Related Words: BSORT 2SORT ADDR.EXCH?

BSOUT (-- addr) "b s out"

User variable containing the character used to output a backspace. Default is 08 hex.

Useful for working with different terminals.

Related Words: BSIN

BSR-CODE (-- 6100 , in HEX)

Constant equal to the machine language 68000 opcode for BSR. BSR op code = 61XX hex where XX = displacement. If XX = 00 hex then the 16 bit word following BSR is used for a displacement value. BSR = Branch Subroutine.

Used internally for 68000 machine coding.

Related Words: DIDCODE JSR-CODE RTS-CODE

BYE (--)

BYE will exit JForth, returning the memory to the system.

As long as the programmer has utilized the JForth-supplied words for memory allocations, opening & closing files, and opening libraries, JForth will return any of these resources which are still pending.

NOTE: Only the standard predefined libraries are cleaned-up so the programmer should take care to close any custom libraries defined with :LIBRARY.

Related Words: AUTO.TERM

BYTE (<name> --)

Define a byte wide structure member. See section on Amiga 'C' Structures.

BYTE-SWAP (n -- ns)

Swaps the lower byte-pair of n.

```
HEX 1234 BYTE-SWAP . ( print 3412 )
```

Related Words: SWAP WORD-SWAP

! " # $ % & ` ( ) * + ' - . / 0-9 : ; < = > ? @ AZ [\ ] ^ _ az { | } ~

C! (n addr --) "c store"

Store the lower 8 bits of N at address ADDR in memory. The higher bits are discarded. Used for storing ASCII characters or other 8 bit values. Odd addresses can be used with this word.

Related Words: C@ ABSC! W! !

C@ (addr -- n) "c fetch"

Fetch a byte from address ADDR in memory and leave it on the stack as a 32 bit value with the high 24 bits set to zero. Usually used for accessing ASCII characters and other 8 bit values

Related Words: C! @ ABSC@ W@

CALL (various-params <library> <routine> -- result)

Call an Amiga Library routine. CALL is an IMMEDIATE word, which can ONLY be used in a colon definition. At compile time it will look up the calling parameters in the FD file and build the appropriate code that, at run time, will:

1) Load the correct registers in order from the stack. (The order the arguments appear in source text is the same as listed in the AmigaDOS technical manuals, and with the ARGS command.)

2) Load Amiga register A6 with the correct library base pointer.

3) Call the correct location, offset from the base pointer.

4) Use register D0 as a 'universal' return code, moving it to top of stack.

Please note that addresses passed to the Amiga must be converted from RELATIVE to ABSOLUTE! Please see the words >ABS and CALL>ABS for this. Please also see the chapter on Calling Amiga Library Routines for more information about CALL.

Related Words: CALLVOID CALL>ABS AREGS>ABS :LIBRARY DCALL >REL >ABS

CALL>ABS (various-params <lib> <routine> -- result)

Call an Amiga Library Routine. This differs from CALL by automatically converting any parameters passed in an address register to absolute before calling. NULL values are left untouched. This saves you from having to use IF>ABS . Use ARGS to see which parameters are passed in address registers. Generally routines in GRAPHICS, INTIUITION and EXEC work with CALL>ABS while routines in the DOS library do not because they pass addresses in data registers!

Please see the chapter on Calling Amiga Library Routines for more information about CALL.

Related Words: CALL AREGS>ABS ARGS CALLVOID

CALLADR, (addr-of-code --) "call address comma"

CALLADR, will cause the compiler to generate, at HERE, an optimized indirect reference to the address on the stack. The address MUST contain executable code.

The indirect reference will be assembled in whichever of the following addressing modes is both possible and most efficient:

16-bit relative (BSR).

```
Address-register indirect with offset (JSR).
Long absolute  (JSR).
```

Refer to the Motorola 68000 Technical Manual for more information on addressing modes.

Related Words: CFA, HERE [ ]

CALLVOID (various-parameters <library> <routine> --)

Call an Amiga Library routine without returning a result. As an example the DOS library Delay() routine does not have a result. Here is an example of calling it.

```
: DELAY()  ( n -- , delay for N/50 seconds )
    CALLVOID DOS_LIB DELAY ;
100 DELAY() ( wait two seconds )
```

Please see the chapter on Calling Amiga Library Routines for more information about CALL.

Related Words: CALL AREGS>ABS ARGS CALL>ABS

CANCELKEY? (-- n)

See the chapter on CLONE.

CANCELNOW? (--)

See the chapter on CLONE.

CARRAY (numbytes <name> --) "c array"

Same as ARRAY except that the size of each element is one byte. Creates 1 dimensional array of length N byte-sized values starting at next available dictionary location . Bytes are initialized at compile time to zero's. The word that is created will have the following stack diagram:

MY-CARRAY (index -- addr)

Related Words: ALLOT ALLOCBLOCK ARRAY

CASE (n -- n)

CASE is a Forth construct for selecting between several actions based on integer value. It is used as follows:

```
: EXAMPLE   ( N -- )
   CASE   0 OF  ." None at all! "    ENDOF
      " Still checking... "
      1 OF  ." Only one "        ENDOF
      13 19 RANGEOF  ." In the teens!"       ENDOF
      dup 5,000 > ?OF  ." Lots and lots!"  ENDOF
      ." Unexpected number = " DUP . CR
   ENDCASE   ( ENDCASE does a DROP )
;
```

When executed, EXAMPLE will check the top number on the stack against each of the numbers before each of the "OF"'s. If it matches, it will drop the number that was being checked then execute the program after the OF up to the ENDOF , then jump to the code after the ENDCASE . if the number does not match, it skips to the code after the ENDOF , leaving the number still on the stack.

Similar to switch() in 'C'.

Related Words: OF ?OF RANGEOF ENDCASE ENDOF ELSE

CD (<directory-name> --) "c d"

CD is used to change the current directory, an operation not possible with the DOS function. This is a JForth program, given the same name as its AmigaDOS counterpart.

CD is the only AmigaDOS function that is completely redefined in the JForth environment. Other AmigaDOS commands are available via DOS.

NOTE: for memory considerations, the JForth 'CD' does not display the current directory if typed in with no pathname; this can still be done via "DOS CD".

```
CD JU:     ( to actually change directories )
```

Related Words: DOS

CELL (-- 4) "cell"

CELL is a constant equal to the number of bytes in a stack item. JForth uses 32 bit, 4 byte stack items so CELL = 4.

CELL allows for code transportability between Forths of different sizes, i.e. 16 bit vs. 32 bit.

A convenient method for defining CELL is:

```
SP@ SP@ - ABS CONSTANT CELL
```

Related Words: CELL+ CELL- CELLS CELL/

CELL+ (n -- n+cell) "cell plus"

Add CELL to the value on the stack. Use for address incrementing , making transportable code .

Related Words: CELL

CELL- (n -- n-cell) "cell minus"

Subtract CELL from the value on the stack.

Related Words: CELL

CELL/ (n -- n/cell) "cell slash"

Divide the value on the stack by CELL.

Related Words: CELL

CELLS (n -- n*cell) "cells"

Multiply the value on the stack by CELL.

Related Words: CELL

CFA, (cfa --) "c f a comma"

This is a deferred word used by the compiler to compile a reference to a Forth word. See (CFA,) .

CHOOSE (n -- r)

Choose a random number R between 0 and N-1 inclusive.

Handy for games, statistics, random art and music. Only good for N <= 32767.

CHOOSE is only pseudo-random but is good enough for most simple applications. You can even get a pseudo-gaussian distribution if you add up the results of several CHOOSE operations, say 7.

CHOOSE uses the value in RAND-SEED for its seed. CHOOSE will always give you the same "random" sequence for the same seed. In a program where you want a different random sequence every time, try setting RAND-SEED to the current system time when you start the program.

```
INCLUDE? CHOOSE JU:RANDOM
: ROLL.DIE   ( -- 1-6 )
    6 choose 1+
;
```

Related Words: RANDOM WCHOOSE RAND-SEED

CLIST? (--)

```
CLIST_LIB
CLIST_NAME
```

Used to manage the CLIST library. See :LIBRARY.

CLONE (<wordname> --)

Clone is used to generate a Royalty Free executable image from your Forth programs. It is a two-pass program which first creates a call dependency tree for <wordname>, then uses the tree to build a separate program image of minimum size.

See the chapter on CLONE.

CLOSEALLLIBS (--) "close all libs"

This word will close all of the standard Amiga Libraries, except for EXEC and DOS, which are closed when BYE is called.

Note that this word is not usually needed; the preferred programming technique is to use the XXX? and -XXX words, where XXX is the name of the library. See :LIBRARY.

Note also that CLOSEALLLIBS will not close any additional libraries the programmer has defined with :LIBRARY. It is the responsibility of the calling program to insure these custom libraries are closed.

CLOSEALLLIBS is normally only used by BYE .

Standards: JForth unique

Related Words: BYE LIBRARY

CLOSEFILES (addr --)

CLOSEFILES accepts the address of a memory block (gotten via ALLOCBLOCK) which has been used to store a list of opened files, and will close each file still there.

CLOSEFILES will not free the memory block.

CLOSEFVREAD (var-addr --) "close f v read"

CLOSEFVREAD accepts a variable or user-variable address that contains the location of a virtual buffer which has been used for reading ONLY.

The memory-block serving as the buffer will be freed, and the var-address will be cleared.

See chapter on File I/O.

Related Words: OPENFV READLINE

CLOSEFVWRITE (file-pointer var-addr --)

CLOSEFVWRITE accepts a variable or user-variable address that contains the location of a virtual buffer which has been used for writing, and the file-pointer associated with the buffer.

If the FREEBYTE-counter for the memory-block is non zero indicating data is present, it is written to the file. Then, the memory-block serving as the buffer will be freed, and the var-address will be cleared.

See chapter on File I/O.

Related Words: OPENFV F, FREEBYTE

CLR-BIT (n bit# -- n') "clear bit"

Force a bit in N to zero. BIT# determines which bit is set to zero. BIT# = 0 refers to the least significant bit.

Related Words: SET-BIT

CLRCHAR (-- var-addr) "clear character"

This is a VARIABLE which contains the ASCII value used by CLS to clear the screen.

CLS (--) "c l s"

Clear the screen. Fetch the ASCII value contained in the VARIABLE CLRCHAR, and send it to the standard EMIT device.

Related Words: EMIT ASCII CLRCHAR

CMOVE (source-addr dest-addr count --) "c move"

Move count bytes from source to destination. The move will proceed from low to high address. If the destination region overlaps the source region and the destination is at a higher address you may want to use CMOVE> or MOVE instead to avoid destroying data.

For large amounts of data MOVE is much faster and will handle any overlaps automatically.

Related Words: MOVE CMOVE> $MOVE

CMOVE> (source-addr dest-addr count --) "c move back"

Move COUNT bytes from source to destination. Unlike CMOVE , start with the last byte and work towards the beginning. Useful for overlapping memory areas.

Related Words: MOVE CMOVE

CNT>RANGE (n count -- n+count n) "count to range"

Convert a standard VALUE COUNT argument pair to a LIMIT INDEX argument pair (suitable for entry into a DO-LOOP).

```
: FOO 10 5  CNT>RANGE  DO I . LOOP ;
( will print: 10 11 12 13 14 )
```

CODE (-- , <wordname>)

Invokes the RPN Assembler to create a new word called <wordname>.

See the chapter on 68000 ASSEMBLY.

COLD (--)

Initialize Forth. COLD will reset most of JForth to the same state it was in on startup or when FREEZE was last called. The parts of JForth affected include:

All USER-variables below SPARE (those defined in the kernal)

Vocabulary and dictionary structures.

All MODULEs are DETACHed.

Note that the arrangement of files, memory areas and libraries will not be affected.

Users should be careful not to execute COLD if the system is currently executing definitions (via DEFERred words) which will be purged; the preferred method is to remove these vectors, replacing them with those which exist in the frozen image.

Users can define an AUTO.INIT word to reset their own code when COLD is called. See AUTO.INIT .

Related Words: ABORT COLDEXEC FREEZE

COLDEXEC (--)

COLDEXEC is a DEFERed word; the JForth default contents is NOOP.

COLDEXEC may be set (via IS) to execute a programmer-defined word during COLD, just prior to the call to ABORT. This is called when you first bring up JForth. You can use this to have special initialization automatically occur on startup. It is necessary to FREEZE the image after setting COLDEXEC to have it execute at the next COLD. (This is done automatically by SAVE-FORTH).

```
' MY-INIT-WORD IS COLDEXEC     FREEZE COLD
```

We recommend the use of the new AUTO.INIT facility instead of COLDEXEC.

Related Words: COLD ABORT FREEZE AUTO.INIT

COMPARE (addr1 addr2 #bytes -- result)

Compare two strings, S1 starting at address ADDR1 , the other, S2 starting at address ADDR2 , paying attention to the case of alphabetic characters. The number of bytes to be compared is on top of the stack.

Results are returned as follows:

```
String relationship      Result
```

```
s1 = s2 .............  0
s1 > s2 .............  1
s1 < s2 ............. -1
```

An example that uses a compiled string and a string at PAD follows:

```
: BIGGERTHANLIFE? ( $string -- result )
    COUNT DROP " LIFE" COUNT COMPARE ;
" MARYLIN" BIGGERTHANLIFE? . ( print 1 )
```

Related Words: TEXT=? $= MATCH? SCAN $-

COMMAS (--)

Turn on the use of commas in the outputting of numbers. If enabled, commas occur every 3 digits in decimal and every 4 digits otherwise.

```
12345 . ( prints 12,345 )
```

Related Words: (COMMAS) NO-COMMAS DIGS/,

COMPILE (<name> --)

COMPILE is an immediate, compile-only word, the use of which may be illustrated by a sample definition:

```
VARIABLE USE-32BIT
: COMPILE-SIZED-FETCH  ( -- )
    USE-32BIT @
    IF    COMPILE  @
    ( 4 bytes long, compile a @ )
    ELSE  COMPILE w@
    ( 2 bytes long, compile a w@ )
    THEN  ; IMMEDIATE

VARIABLE VAR1
USE-32BIT OFF
: GETVAL  VAR1 COMPILE-SIZED-FETCH ;
```

Our hypothetical example assumes that we may be compiling for one of two memory systems...one that will need 16-bit fetches when it runs, another at 32 bits. The variable USE-32BIT has been set to the appropriate state for the one we are doing.

The job of the above definition is not to FETCH the correct size at compile-time, but rather to COMPILE the correctly-sized operator to be executed later, when the application actually runs.

Related Words: IMMEDIATE [COMPILE] LITERAL

COMPILING? (-- flag)

Returns a flag reflecting true (if COMPILING) or false (if not COMPILING) based on the value of the variable STATE .

Related Words: STATE ?COMP ?EXEC

CONSOLE?

```
CONSOLE_LIB
CONSOLE_NAME
```

These are used to manage the Amiga console library . See :LIBRARY .

CONSOLE! (file-pointer --) "console store"

This word accepts an AmigaDOS file-pointer, and installs it into the CONSOLEIN and CONSOLEOUT variables, causing the file to become the console through which EMIT and KEY work.

Usually, the file-pointer represents an AmigaDOS console window of either type RAW: CON: or NEWCON:.

CONSOLE@ (-- file-pointer) "console fetch"

This word returns the AmigaDOS file-pointer currently installed as the CONSOLEOUT window.

Usually, the file-pointer represents an AmigaDOS console window of either type RAW: CON: or NEWCON:.

Related Words: EMIT KEY FOPEN

CONSOLEIN (-- var-addr) "console in"

CONSOLEIN is a variable that usually contains the AmigaDOS file-pointer through which KEY will receive characters.

Usually, the file-pointer represents an AmigaDOS console window of either type RAW: CON: or NEWCON:.

Related Words: CONSOLE! CONSOLEOUT CONSOLE@

CONSOLEOUT (-- var-addr) "console out"

CONSOLEOUT is a variable that usually contains the AmigaDOS file-pointer through which EMIT will output characters.

Usually, the file-pointer represents an AmigaDOS console window of either type RAW: CON: or NEWCON:.

Related Words: CONSOLE! CONSOLEIN CONSOLE@

CONSTANT (value <name> --)

Define a Forth word that will return the value when called. The use of constants is recommended whenever possible in your code because it makes it easier to change and easier to understand.

```
123 CONSTANT MYVAL
MYVAL . ( print 123 )
```

Related Words: VALUE VARIABLE CREATE DOES>

CONTEXT (-- addr)

A user-variable containing a pointer to the vocabulary which is first to be searched by FIND. CONTEXT may be set simply by declaring the name of the vocabulary you wish to access.

The function VLIST will display the words in the CONTEXT vocabulary.

See the section on Vocabularies.

Related Words: CURRENT VLIST ALSO PREVIOUS ORDER VOCS FIND WORDS

CONVERT (d1 addr1 -- d2 addr2)

```
d1 = value accumulator
addr1 = address 1 char before numeric string
d2 = d1 + <addr1+1>
addr2 = address of first non-digit
```

Convert a string at address ADDR1+1 to a double number and add it to D1 . Leave the sum D2 on the stack and push the address ADDR2 of the first non-digit found on top of the stack. You can now process the non digit and continue with another call to CONVERT .

Related Words: NUMBER NUMBER?

COUNT ($string -- addr count)

Take a Forth text string and return the address of the first character and the number of characters.

Definition: : COUNT (addr -- addr+1 byte) DUP 1+ SWAP C@ ;

Also used as a C@ with auto increment. See '83 handy reference.

```
" Hello" COUNT TYPE
```

CR (--) carriage return

CR is a DEFERed execution word; see (CR). Transmits a carriage return and line feed to the selected output device . Also flushes the FASTEMIT buffer if FAST I/O mode (line-buffered) is in effect.

Related Words: (CR) CR? >NEWLINE FLUSHEMIT FAST EMIT

CR? (--) carriage return ?

Does a carriage return and line feed if the input text is near the end of a line .

Related Words: CR (CR) OUT >NEWLINE

CREATE (<name> --)

Define a new word in the dictionary.

CREATE creates a "header" structure in the dictionary comprised of:

1) a name field -- built from the next text in the input stream.

2) a link field -- used to tie the headers together for searching. The link field points to the name field of the previous definition.

3) a size field -- used by the compiler to determine compile-time characteristics.

4) a code field -- contains actual 68000 assembly language which determines the run time behavior of the word.

Words that use CREATE are called DEFINING-WORDS. CREATE is used by all defining words, such as CONSTANT , : , VARIABLE , etc... Also used with DOES> to create new data type. See the tutorial for more information and examples.

Related Words: CREATECHAR CONSTANT : VARIABLE DOES> (CREATE) DO-DOES-SIZE SKIP-WORD

CREATECHAR (-- var-addr)

This is a user-variable that holds the character that is used to parse the input stream when a dictionary name-field is being CREATEd. This character is used as a delimiter.

Related Words: CREATE

CSP (-- addr) "c s p"

Variable used by many of the compiling words to check the stack depth. !CSP will set CSP to the current SP value; ?CSP will later verify CSP as unchanged, or generate an error. SP = stack pointer.

Related Words: CSP! ?CSP DEPTH

CURRENT (-- addr)

CURRENT is a variable containing a pointer to the vocabulary to which newly-created words will be attached. It is set to point to a specific vocabulary by executing that vocabulary name, followed by DEFINITIONS.

Related Words: ALSO ONLY VOCABULARY PREVIOUS ORDER VOCS FORTH ABORT COLD CONTEXT

D! (d addr --) "d store"

Store the 64 bit double number D at memory address ADDR. ADDR must be even. Use ODDD! for odd addresses.

```
: DVARIABLE ( name -- , define double variable )
   VARIABLE CELL ALLOT ;
DVARIABLE DVAR1
$ 12345678.32415762 DVAR1 D!
DVAR1 D@ D.
```

Related Words: D@ D+ D- 2@ 2!

D+ (d1 d2 -- d1+d2) "d plus"

Add two 64 bit double numbers.

D- (d1 d2 -- d1-d2) "d minus"

Subtract top double number from second double number.

D. (d --) "d dot"

Output the ASCII string for a double number. Utilizes JForth COMMAS mode (commas inserted every 3 digits in decimal; 4 digits in other bases). See COMMAS.

```
4567884. D. ( prints 4,567,884 )
```

Related Words: COMMAS NOCOMMAS . D.R .R U. U.R

D.R (d field-size --) "d dot r"

Print a double number D right justified in a field of FIELD-SIZE characters.

```
12345678.90  12 D.R
```

Related Words: D. . COMMAS NO-COMMAS .R U.R U.

D2* (d -- d*2) "d two times"

Double number fast multiply by 2.

Related Words: 2* *

D2/ (d -- d/2) "d two slash"

Double number fast divide by 2.

```
10.  D2/  D. ( will print out 5 )
```

Related Words: D2*

D< (d1 d2 -- flag) "d less than"

Compare two double numbers. Return true if D1 less than D2.

D= (d1 d2 -- flag) "d equals"

Compare two double numbers. Return true if D1 equals D2.

D@ (addr -- d) "d fetch"

Fetch the double number D from even address ADDR.

Related Words: ODDD@ @ C@ W@ 2@

DABS (d -- |d|) "d absolute"

Calculate absolute value of D.

Related Words: ABS

DCALL (various-params <library> <routine> -- D0 D1)

DCALL is functionally identical with CALL, except it returns both D0 and D1 (-- D0 D1) as a double number.

Double number results are returned by the MATHIEEEDOUBBAS library. See the chapter on Calling Amiga Libraries

Related Words: CALL

DDROP (d --) "d drop"

Discards the double number D that was on top of the stack.

Related Words: 2DROP DROP RDROP

DDUP (d -- d d) "d dup"

Duplicates the top stack item d .

Related Words: DUP 2DUP

DECIMAL (--)

Set numeric base to decimal by setting BASE to 10.

```
HEX 100 DECIMAL . ( prints 256 )
```

Related Words: HEX BINARY BASE

DEF (<name> --)

DEF will disassemble a given Forth word in Motorola format. See the chapter '68000 Assembly'.

Related Words: WORDS-LIKE DISM ADISM DUMP '

DEFER (<name> --)

Create a vectored word.

DEFER creates a function that has a user variable area storage cell containing a pointer to the word it will execute when it is called. It is a vectored execution word. Use IS to change what a deferred word will execute. Use WHAT's to get the CFA of what a deferred word will execute.

```
DEFER EXAMPLE
EXAMPLE  ( will print nothing, calls QUIT )
' ORDER IS EXAMPLE  ( will set EXAMPLE  to execute ORDER )
EXAMPLE  ( will execute ORDER )
```

```
WHAT'S EXAMPLE  ( will leave the CFA of ORDER )
>NAME ID.       ( prints ORDER )
```

See the section on DEFER under Forth Tools.

Related Words: WHAT'S IS GLOBAL-DEFER

DEPTH (-- #of-cells-on-stack)

Counts the number of cells on the stack . Places the count on top of the stack .

```
: CHECK-OPERATION  ( -- , monitor stack change )
    DEPTH >R  DO-OPERATION
    DEPTH R> - ABS  ( calculate change )
    ." Stack depth changed by " . CR ;
```

Related Words: US-DEPTH RDEPTH .S 0SP SP@

DETACHMODULE (-- , <modulename>)

See the chapter on MODULES.

DICTIONARYSIZE (-- var-addr)

See the chapter on CLONE.

DIGIT (char base -- digit true | char false)

DIGIT attempts to convert the ASCII character CHAR to a number according to the given numeric BASE.

If successful, it returns the converted digit and a TRUE, otherwise it leaves the unaffected CHAR and a FALSE.

```
DECIMAL  ASCII 9  10  DIGIT SWAP . .  ( print 9 -1 )
ASCII A  10  DIGIT SWAP . . ( failed, print 65 0 )
```

Related Words: (NUMBER)

DISKFONT?

```
DISKFONT_LIB
DISKFONT_NAME
```

Used to manage the Amiga DISKFONT library. See :LIBRARY .

DLITERAL "d literal"

Compile time: (d --)

Run time: (-- d)

Compile a literal double number. DLITERAL is an immediate word that takes a double number from the stack, and compiles it into the dictionary following DLIT . When this code is executed, the number will be pushed to the stack. DLITERAL is similar to LITERAL which compiles a single precision number.

```
: D64K+   ( d -- d+64K , add 64K to double number on stack )
\ calculate double 64K at compile time
    [ DECIMAL 64 1024 * S->D ]
```

```
            DLITERAL     \ compile DLIT and the double number
            D+ ;         \ and, at run time, add it to whatever
```

DNEGATE (d -- -d) "d negate"

Negate the value of a double number.

The bitwise logical operation is -1 XOR 1+

```
3.4 DNEGATE D.  ( prints -34 )
```

Related Words: NEGATE

DO (limit index --)

DO marks the beginning of a DO....LOOP instruction.

If, at run time the limit is not larger than the index, DO will NOT execute the body of the loop; instead it will drop the terms, and jump past the corresponding LOOP.

Standards: 83. '79 and fig are slightly different, executing the body of the DO-LOOP at least once, regardless of limit-index relationship.

```
: PRINT-0-9  ( -- )  10 0  DO  I .  LOOP  ;
```

Related Words: LOOP -LOOP +LOOP LOOP-DROP

DO-DOES-SIZE (-- N) "do does size"

DO-DOES-SIZE is a constant equal to the number of bytes between the CFA of a CREATE word and the data area. It is used to define >BODY and BODY> .

Related Words: >BODY DOES> >PARENT VLINK> CREATE CONSTANT

DOES> (-- addr , at run time)

DOES> is used with CREATE to create new defining words in Forth. The code following CREATE determines how the new word is created. The code following DOES> determines what the new word does.

We could define a simple version of constant as:

```
: CONSTANT CREATE , ( save number in dictionary )
   DOES> @ ( fetch saved value ) ;
73 CONSTANT CON1  ( execute CREATE portion )
CON1 .  ( execute DOES> portion then print 73 )
```

Suppose we want to CREATE a word that contains an offset supplied on the stack at compile time. This word should add the offset to a number on the stack at run time. This is a simplified version of the structure members used for accessing Amiga 'C' structures. Here is how that code would be produced.

```
: ADDER  ( offset -- , define new kind of Forth word )
   CREATE   ( offset -- , how to make )
      ,  ( save offset in dictionary )
   DOES>    ( n addr -- value+offset , what to do )
      @ + ( fetch saved offset and add it to N )
;
20 ADDER ADD20  ( n -- n+20 , make one )
```

```
12 ADDER DOZENMORE   ( n -- n+12 )
100 ADD20 .  ( now do it, print 120 )
100 DOZENMORE . ( print 112 )
```

Related Words: CREATE IMMEDIATE

DOLINES (<filename> --)

Process each line of a file using a deferred word. See the chapter on file I/O.

DOS (<command-line> --) "dos"

This word will parse the rest of the current line of the input stream, submitting it to AmigaDOS as a CLI command line.

Virtually any AmigaDOS command may be invoked, with the following limitations:

1. AmigaDOS output will appear in the JForth window, but cannot be parsed or interrupted. Generally, AmigaDOS commands that produce long streams of text are better executed from a CLI. The DOS interface is primarily provided for task and file control.

2. Commands executed via DOS are executed in their own environment which terminates when the command ends. Therefore, any AmigaDOS command that changes only the local environment (like CD) will have no effect. JForth provides its own CD for changing directories. See CD.

```
DOS COPY MYFILE RAM:
DOS DATE
DOS CD   ( will print the current directory )
```

3. DOS will look for the command in the C: directory. It will not use the search path. If you want it to look in a directory in RAM: instead of on the Workbench disk, you must assign C: to your directory in RAM.

4. DOS cannot be used in a colon definition because it is not IMMEDIATE. Use $DOS instead.

Related Words: $DOS DOSCOMMAND CD

DOS0 (-- addr) "dos zero"

DOS0 returns the address of the string buffer used for converting forth-type strings to AmigaDOS compatible null-terminated strings.

The DOS0 buffer resides in the user-area, and is 256 characters long. A count byte is maintained 1 byte below DOS0.

Related Words: >DOS +DOS

DOS? (--) "dos question"

```
DOS_NAME   dos name
DOS_LIB    dos lib
```

Used to manage the DOS library. See :LIBRARY. DOS is opened and closed automatically by JForth.

DP (-- addr) "d p"

A user-variable used to hold the next available location in the dictionary for this user.

! " # $ % & ` ( ) * + ' - . / 0-9 : ; < = > ? @ AZ [\ ] ^ _ az { | } ~

DP is referenced by several high-level words, and is therefore not generally needed by the programmer. The contents of DP are put on the stack by HERE , and incremented by ALLOT.

Whenever the system is INTERPRETing, the contents of DP MUST BE WORD-ALIGNED! See ALIGN.

Standards: fig '79 '83

Related Words: ALLOT HERE ALIGN

DPLIMIT (-- addr) "d p limit"

User variable that holds the maximum value allowed for dictionary to grow to. Checked by ? STACK . The data stack and the dictionary occupy the same memory area. The stack starts at the top and grows down. The dictionary starts at the bottom and grows up.

Standards: JForth

Related Words: ?STACK DP

DPL (-- addr) "d p l"

User variable that shows how many digits are to the right of a decimal point when converting a character string to a number . When no decimal point is seen , NUMBER leaves -1 in DPL .

Standards: fig '79 '83

```
23.45 DPL ?  ( prints 2 )
```

Related Words: USER INTERPRET NUMBER (NUMBER)

DROP (n --)

DROP discards the top value N on the stack.

Standards: fig '79 '83

```
DUP DROP  ( cancel each other out )
```

Related Words: DDROP 2DROP RDROP XDROP

DST (addr <structure> --) "dump struct"

Dump a structure at address ADDR using the named structure as a template. DST will display the contents of each member, whether it is signed or unsigned, its width in bytes, and its name.

```
GETMODULE INCLUDES
BITMAP MYBMAP
3 MYBMAP ..! BM_DEPTH
MYBMAP DST BITMAP  ( display contents )
```

Related Words: DUMP ..@ FILE?

DSWAP (d1 d2 -- d2 d1) "d swap"

```
or ( a b c d -- c d a b )
```

Reverses the top two pairs of numbers on the stack. This can be used to swap two double precision numbers.

Standards: fig '79 '83, sometimes called 2SWAP. Both are allowed in JForth.

Related Words: SWAP 2SWAP DDUP >R

DU2* (ud1 -- ud1*2) "d u 2 times"

Left shift the unsigned double number UD1. MSB is shifted out and discarded. 0 is shifted in to LSB. This can be used as a fast multiply by 2 or as a shift.

Standards: JForth unique

Related Words: * U* 2* U2* M*

DU2/ (ud1 -- ud1/2) "d u 2 slash"

Right shift the unsigned double number UD1. 0 is shifted in to MSB . LSB is shifted out and discarded. This can be used as a fast divide by 2.

Standards: JForth unique.

Related Words: / */ u/ U2/

DU< (d1 d2 -- flag) "d u less than"

Flag = TRUE if D1 is less than D2, FALSE otherwise. Same as U< only for double numbers.

Standards: '83

Related Words: < > U<

DUMP (addr u --)

Display memory from ADDR for U bytes on the standard EMIT device.

Each line displays 16 bytes in hexadecimal and ASCII forms; any control/non-printing characters in the ASCII columns will be displayed as a period . The CONSOLE window should be set to the full screen width.

Every 23 lines, a header is printed showing the low nibble value of the address for that column above both formats.

DUMP allows the programmer to suspend the output; see ?PAUSE.

Related Words: EMIT ?PAUSE

DUP (n -- n n) "doop"

Duplicates the top item on the stack.

Standards: fig '79 '83

Related Words: 2DUP DDUP -DUP DUP>R ?DUP

DUP>R (n -- n) (--R-- n) "doop to r"

Push a copy of N onto the return stack. This is simply a faster way to do DUP >R

Related Words: DUP ?DUP >R

ECHO (-- addr)

If this variable is TRUE, then INCLUDE will echo each line it compiles to the screen.

ELSE (--)

ELSE is used with IF and THEN to specify the code to execute if the conditional was false.

```
: PRINT-ACCOUNT-STATUS   ( -- , good or bad news )
   ." Your account is in the "
   ACCOUNT-BALANCE @  0>
   IF   ." BLACK."
   ELSE ." RED"
   THEN ;
```

The above example uses the convention of putting related IF ELSE and THEN words at the same level of indentation.

Standards: FIG '79 '83. However, the specifics of the JForth implementation is fig. The fig models provided the most error detection and also give the most information.

Related Words: IF THEN BEGIN WHILE UNTIL BRANCH

EMIT (char --)

EMIT is a DEFERed word, see (EMIT).

The basic function of this word is to make the ASCII character appear on an output device, usually the system CONSOLE.

```
ASCII X EMIT   ( Prints X )
```

Related Words: TYPE DUMP CR SPACES R D

EMIT-TO-COLUMN (char column# --)

Emit CHAR to the standard EMIT device, until column# is reached by the cursor. This word is useful for producing columnized, tabled lists.

```
: 1LINER ( page# $item -- , connect item and page )
    CR $TYPE  ( draw item )
    ASCII . 40 EMIT-TO-COLUMN  ( draw bar )
    4 .R  ( page number )
;
17 " Chapter 2" 1LINER
234 " Chapter 5" 1LINER
```

Related Words: EMIT OUT >NEWLINE CR?

ENABLE_CANCEL (-- addr)

See the chapter on CLONE.

END (--)

Synonym for UNTIL, see UNTIL.

Standards: fig '79 '83 .

Related Words: UNTIL

END-CODE (--)

See the chapter on 68000 ASSEMBLY.

ENDCASE (n --)

ENDCASE is the word used to end a CASE statement. See CASE. ENDCASE is an immediate word that resolves all of the CASE forward branches.

Related Words: CASE OF ENDOF =

ENDOF (--)

ENDOF is a word used in a CASE statement. See CASE.

EOL (-- eol) "e o l" or "end of line"

EOL is a CONSTANT, and returns the value recognized by the Amiga CONSOLE device as a End-Of-Line. You will also find this value used in files at the end of lines.

UNIX and Amiga DOS use $0A, or "linefeed" as the EOL character. The Macintosh, from Apple Computer, uses a $0D or "return" as an EOL.

Related Words: CR FEMIT READLINE >NEWLINE

EOF (-- EOF) "e o f" or "end of file"

EOF is a constant returned by FKEY when an end of file has been reached.

Related Words: FKEY

ERASE (addr count --)

Set COUNT bytes beginning with ADDR to zero.

Standard note: JForth

```
PAD 256 ERASE  ( clear 256 bytes at PAD )
```

Related Words: FILL CMOVE MOVE DO

ERROR (error# --)

When executed, ERROR will:

1. Go to a new-line, if not already there.
2. Print the text currently at HERE (that which was last INTERPRETed) followed by a '?'.
3. If error# is non-zero, print "Message # " and the error number.
4. Clear the data stack and execute QUIT.

ERROR is the standard error-exit routine for the compiler/interpreter, which passes 0 for the error number.

The use of numbers for error codes is discouraged in JForth; the suggested method is to provide a full-text error message via ?ABORT" , .ERR , or your own error handler, executing QUIT if necessary.

Standards: fig. '79 and '83 use ABORT and ABORT"

```
: IN-DICTIONARY?  ( -- cfa , QUITS with ? if not there )
    BL WORD FIND NOT
    IF 0 ERROR
    THEN ;
```

Related Words: ABORT ABORT" ?ERROR .ERR HERE QUIT

ERRORCLEANUP (--)

See the chapter on CLONE.

EVEN-UP (n -- n+1 | n)

Increment N if odd. N remains the same if even. If N is odd, add 1 to make it even.

Standards: JForth unique.

Related Words: ALIGN AND

EXEC? (--)

```
EXEC_LIB
EXEC_NAME
```

Used to manage the Amiga exec library. See :Library. The Exec library is opened automatically by JForth.

EXECUTE (cfa --)

EXECUTE causes immediate execution of the word whose cfa (code-field-address) is on the stack. Execution then continues at the next instruction following the call to EXECUTE . EXECUTE uses a 68000 Jump Subroutine instruction.

Standards: '79 '83.

```
: HI ." HELLO" ;
' HI .S EXECUTE


THE-ACTION @  ACTIONS-ARRAY @  EXECUTE   ( jump table )
```

Please note!!! If you use a "jump table" in a cloned program, you must initialize the program at run time! See Clone.

Related Words: @EXECUTE '

EXISTS? (<name> -- exists-flag) "exists question"

EXISTS? returns a TRUE if the following word is in the dictionary.

```
EXISTS? GR.INIT .IF ." Graphics loaded!" .THEN
```

Related Words: FIND .NEED ' .IF

EXIT (--)

Exits from a colon definition by executing an RTS, Return from Subroutine, instruction. Don't use inside DO loops .

Standards: '79 '83 . Called ;S in fig.

Related Words: UNTIL WHILE RETURN LEAVE

EXPECT (addr maxchars --)

Used for inputting a string. Read up to MAXCHARS characters from current KEY device to address starting at ADDR. EXPECT will echo characters as they are entered. EXPECT will stop getting characters when the maximum is reached or when a Carriage Return key is hit. EXPECT handles Backspace characters and Control-X. The number of characters read will be available in the user-variable SPAN following the call.

The Command Line History system works by installing a special version of EXPECT. Use HISTORY.OFF to get vanilla traditional version.

When using EXPECT in cloned programs, set the variable RAWEXPECTECHO if you are using RAW: windows and need EXPECT to echo characters as they are typed.

Standards: fig '79 '83

```
: GET-NAME  ( -- , put name at PAD )
    CR ." What is your name? "
    PAD 80 EXPECT
    CR ." Hello " PAD SPAN @ TYPE
;
GET-NAME
```

Related Words: SPAN QUERY KEY

F, (file-pointer var-addr n --) "f comma"

Send the cell N to the specified file at its current position using buffered I/O mode, the buffer address being held in the VARIABLE or USER-variable VAR-ADDR .

The 1024-byte buffer is created by a previous call to OPENFV which also initializes the VAR-ADDR . Buffers allocated for purposes of writing should be closed with CLOSEFVWRITE .

See the chapter on JForth File I/O for more information on virtual file usage.

```
( store DATA in next buffer cell )
MYFILE @ MYBUFFER  DATA F,
```

Related Words: OPENFV CLOSEFVWRITE FFLUSH?

F@, (file-pointer -- cell) "f fetch comma"

Fetch the next sequential 32-bit-value from a file; the cell will be read from the current location for the file. F@, calls FREAD; the programmer should check FERROR after to insure the value returned is valid.

See chapter on File I/O.

Related Words: @ FREAD FERROR F@,?

F@,? (file-pointer -- cell) "f fetch comma question"

This word is functionally identical to F@, except it will print an error message and execute QUIT if an error occurs during the FREAD (calls FREAD?).

JForth provides for optional automatic cleanup of opened files if an error forces a QUIT. See MARKFCLOSE and UNMARKFCLOSE.

Related Words: F@,

FALSE (-- 0)

FALSE is a constant which returns the value of a logical false. Usually zero in most systems.

Related Words: TRUE 0= IF

FAST (--)

After FAST has been executed, the EMIT I/O will be in line-buffered mode, a highly-efficient alternative to the single-character I/O mode. This is the default mode of CONSOLE I/O as JForth is distributed.

In FAST mode, any characters EMITted are held in a buffer until either:

1) The buffer is full.

2) CR , CR? , or FLUSHEMIT is executed.

3) Input is requested by KEY , EXPECT .

FAST acts to install its own vectors in the EMIT vector.

Any characters EMITted in FAST mode will update OUT (via +OUT) to reflect the current cursor column number.

Related Words: SLOW <FASTEMIT> CONSOLE EMIT KEY FLUSHEMIT

FBLK (-- addr) "f b l k"

This is a USER-variable, used to hold the file-pointer for the ASCII file currently being INCLUDEd.

FBLK should not normally be altered by the programmer.

Related Words: INCLUDE

FCLOSE (file-pointer --) "f close"

FCLOSE will close the specified file. Files should be closed when you are through using them. When the user executes BYE, any open files will be closed.

See the chapter on File I/O.

```
MYFILE @ FCLOSE
```

Related Words: FOPEN $FOPEN

FCLOSEVAR (var-addr --)

If you use a variable to hold the address of a file pointer, this will close the file then clear the variable. It first checks to make sure the variable does not contain zero.

```
VARIABLE MY-FILEID
NEW FOPEN RAM:TESTFILE
MY-FILEID !  \ save pointer in variable
MY-FILEID FCLOSEVAR  \ now close it if open
MY-FILEID FCLOSEVAR  \ yes, this is safe
```

Related words: FCLOSE FREEVAR

FENCE (-- addr)

FENCE is a user-variable which may be set to an address below which FORGET will issue a warning.

FENCE may be set by the programmer, but is automatically set by the JForth System Manager to the current HERE in the following situations:

1) By FREEZE (executed by SAVE-FORTH).

2) If FORGET will lower the dictionary below the currently frozen FENCE.

```
( cautions if FORGETting below AARDVARK )
' AARDVARK FENCE !
```

Related Words: FORGET FREEZE SAVE-FORTH

FERROR (-- var-addr) "f error"

FERROR is a user variable which provides the programmer an alternate means (other than return codes) for checking for file operation errors.

Each of the 3 primary file system primitives, FREAD, FWRITE and FSEEK clear FERROR at their start, and adjust it upon completion; if used, it should be checked as soon as the call returns.

```
MYFILE @  PAD  256 FREAD
```

```
    FERROR @ abort" Error in file read"
```

Related Words: FREAD FWRITE FSEEK

FFLUSH? (file-pointer var-addr --)

Flush the contents of the virtual buffer to disk. The variable should contain the address of a buffer which must have been opened by OPENFV .

See chapter on File I/O

Related Words: OPENFV CLOSEFVWRITE F,

FIG-NOT (n -- 1 | 0) "fig not"

If n=0 then leave 1 on stack otherwise leave a zero.

Standard: FIG

Related Words: NOT 0= IF

FILE? (<name> --) "file question"

FILE? is used to identify the file that a word was compiled from, and optionally allows the programmer to search that file for instances of the same text.

In order for FILE? to work for the words from a given file, that file had to have been compiled with FILEHEADERS enabled. This is the default state for JForth as distributed. When FILEHEADERS contains a non-zero value, special dictionary entries are created marking the beginning and end of the file. (4 colons are appended to the name, i.e. ::::MYFILE, to mark the beginning; the end is noted by 3 semi-colons...;;;).

Once the file has been identified and opened, its text is searched for instances of the word's name. If found, that line will be EMITted along with each following non-blank line. The search for another instance will resume when an empty line is encountered.

As the source is not supplied for words below TASK, this utility cannot be used for kernal primitive words, approx. the first 30K of the image. These filenames begin with J and end with either .I or .ASM . JCOMPILER.I and JKERNAL.ASM are examples of these files.

```
    FILE? :STRUCT
```

Related Words: FILEHEADERS EACH.FILE?

FILEHEADERS (-- addr)

This user-variable is examined by INCLUDE to see if the programmer desires file headers, those required for the operation of FILE? . You can save some space in the dictionary by turning FILEHEADERS OFF but then FILE? won't work.

FILEWORD (<filename> -- addr)

FILEWORD operates similarly to WORD, in that it parses the input stream, moving suitable text to HERE, however, it uses parsing rules that allow a filename with spaces to be passed to AmigaDOS.

If the first word that it sees starts with a double-quote "" then it will parse up to the next double-quote, skipping over any spaces.

```
    : FOPEN  FILEWORD $FOPEN ;
```

```
FOPEN "df1:icons are silly"
```

The resultant string may be subsequently null-terminated and moved to the DOS0 buffer with: (-- addr) COUNT >DOS

Related Words: COUNT >DOS WORD HERE FOPEN $FOPEN

FILL (addr u byte --)

Fill U bytes of memory starting at ADDR with BYTE .

```
PAD 256 BL FILL   ( put 256 blanks at PAD )
```

Related Words: C! ERASE ! MEM! CMOVE

FIND ($addr -- $addr 0 | cfa -1 | cfa 1)

FIND searches the CONTEXT vocabulary for the string at $addr, and returns whether it was found, and if so, whether it is an IMMEDIATE word.

0 means the word was **not** found. The original name address will be returned.

If the word was found. A 1 or a -1 and the word's CFA is returned.

1 means the words is IMMEDIATE. INTERPRET uses this when compiling to decide whether to immediately execute the word it finds.

-1 means the word is **not** IMMEDIATE.

If the search is not successful, AND the user-variable SEARCH-CURRENT is non-zero, the search will also be performed in the CURRENT vocabulary.

```
: SHOWDEF  ( <name> -- , disassemble word if found )
    BL WORD  FIND 0=
    IF $TYPE ."  could not be found!" CR QUIT
    THEN DISM ;
SHOWDEF DUP
```

Related Words: WORDS-LIKE FILE? FORGET

FIND-DATA (addr count n --)

Beginning with ADDR , search for COUNT bytes, looking for the 32-bit-value N. Note that ADDR MUST be WORD-aligned, and that only such addresses are checked. The addresses containing matching cells are printed on the standard EMIT device.

Related Words: SCAN

FIND-WDATA (addr count w --)

This is functionally identical to FIND-DATA, except that it searches for WORD-sized data. See FIND-DATA.

FLD (-- addr)

User variable for controlling field length in numeric output.

FLUSHEMIT (--)

FLUSHEMIT is used in FAST mode to force the EMIT buffer to be EMITted even though it is not full, and an end-of-line character has not yet been received.

Related Words: FAST SLOW EMIT

FOPEN (<filename> -- file-pointer | 0) "f open"

Open a file by name. Return zero if file could not be opened. The programmer should ALWAYS check the returned value from FOPEN.

Please see the chapter on File I/O for details.

```
FOPEN RAM:MYFILE  ( open existing file )
DUP 0= ABORT" File could not be opened!"

NEW FOPEN RAM:HOTNEWS  ( open new file )
```

Related Words: FILEWORD DOS0 NEW HERE FREAD FWRITE 0FOPEN $FOPEN FCLOSE

FORGET (--)

FORGET deletes definitions from the dictionary . The specified definition and all definitions following up to the end of the dictionary are forgotten.

If you would like the contents of a file to automatically be forgotten when you REinclude the file, use ANEW which calls FORGET.

If you would like a word to be called if something is forgotten use IF.FORGOTTEN. This can be important if you are forgetting code that has allocated memory or opened files, etc. and want to clean up before losing your pointers.

If you need to redefine what FORGET does, define a word called [FORGET] that calls [FORGET] . This will get called by FORGET. See [FORGET].

```
: FOO ." Hello" cr ;
: GOO 23 + . ;
FORGET FOO
FOO ( won't find FOO or GOO )
```

Related Words: ANEW IF.FORGOTTEN [FORGET] FIND WORDS

FREAD (file-pointer addr max-to-read -- #bytes | -1)

READ will read from the specified file at its current location, placing the contents at RELADDR. FREAD will read until MAX-TO-READ bytes have been read, or end-of-file, whichever occurs first.

FREAD always returns a value. If the call was successful, it returns the number of bytes #BYTES that were read in If an error occurred, it returns -1.

An error is also indicated by a non-zero value in the user-variable FERROR, immediately following the call.

See chapter on File I/O.

```
MYFILE @ PAD 256 FREAD
```

```
PAD SWAP DUMP  ( dump stuff read )
```

Related Words: FOPEN 0FOPEN FSEEK FREAD?

FREAD? (file-pointer addr max-to-read -- #bytes)

FREAD? is functionally identical with FREAD with one exception; if an error occurs during the read operation, FREAD? will print "Error on file read!" and execute QUIT.

JForth provides for optional automatic cleanup of opened files if an error forces a QUIT. See MARKFCLOSE and UNMARKFCLOSE.

For further information, see FREAD.

```
MYFILE @  PAD  256  FREAD? .
```

Related Words: FREAD QUIT MARKFCLOSE UNMARKFCLOSE

FREEBLOCK (memblock --)

FREEBLOCK is used to return memory to the Amiga memory manager that had been previously allocated with ALLOCBLOCK. It performs housekeeping functions along with the JForth Memory Manager and should therefore be done ONCE AND ONLY ONCE for each previous ALLOCBLOCK call.

At BYE, The JForth Memory Manager will return all memory-blocks allocated with ALLOCBLOCK that had NOT been returned via FREEBLOCK. JForth also provides for automatic cleanup of allocated memory areas if an error forces a QUIT. See MARKFREEBLOCK and UNMARKFREEBLOCK.

See section on Memory Management.

Related Words: ALLOCBLOCK MARKFREEBLOCK UNMARKFREEBLOCK FREEVAR

FREEBLOCKS (memory-block --)

FREEBLOCKS accepts the address of a memory block previously allocated via ALLOCBLOCK, which contains a list of other memory blocks.

FREEBLOCKS will proceed through such a list, closing each memory block. The memory block used to store the list will not be freed.

Related Words: FREEBLOCK @&FREEBLOCKS

FREEBYTE (memory-block -- contents-of-counter)

A 32-bit storage location is available for each memory block allocated; the contents of which are returned by FREEBYTE. This is used to keep track of data when the memory block is used as a stack or a virtual file buffer.

The FREEBYTE-counter is initialized to zero when the memory-block is allocated, and update automatically by PUSH POP +STACK -STACK FVREAD and FVWRITE.

See section on Memory Management for details.

```
MYSTACK @  FREEBYTE  CELL/  ( #cells on MYSTACK )
```

Related Words: FREEBYTEA PUSH POP +STACK -STACK

FREEBYTEA (memory-block -- address-of-counter)

FREEBYTEA returns the address of the FREEBYTE-counter for a memory-block. This is particularly useful if the programmer is using the FREEBYTE-counter for custom purposes and wishes to update its value.

```
0  MYMEMBLK @  FREEBYTEA  !
( now FREEBYTE will return 0 )
```

FREECELL (memory-block -- freecell)

This is functionally identical to FREEBYTE , except that it returns the offset of the next free CELL, based on the FREEBYTE-counter.

FREEVAR (var-addr --)

If you use a variable to hold the address of allocated memory, this will free the memory then clear the variable. It first checks to make sure the variable does not contain zero.

```
VARIABLE BIG-BUFFER
MEMF_CLEAR  2000 ALLOCBLOCK
BIG-BUFFER !  \ save pointer in variable
BIG-BUFFER FREEVAR  \ now free it if allocated
BIG-BUFFER FREEVAR  \ yes, this is safe
```

Related words: FREEBLOCK FCLOSEVAR

FREEZE (--)

FREEZE is used to snapshot important JForth system variables which, at COLD-start time, define:

1) The most recently defined word in the dictionary, and the resulting VOCABULARY structure.

2) The vectors for the following DEFERed system words:

```
EMIT        KEY         ?TERMINAL   CR          QUIT
INTERPRET   FIND        :CREATE     NUMBER      ID.
SOURCE      BLOCK       'WORD       LONGCFA,    CFA,
FLUSHEMIT   COLDEXEC    ABORT
```

3) All USER-variables defined below TASK.

Even though the dictionary may be extended, and the above vectors altered by the programmer, their contents at the last FREEZE will be restored by COLD.

FREEZE is called automatically by SAVE-FORTH or if FORGET lowers the dictionary below that saved by the last 'FREEZE'.

Related Words: SAVE-FORTH FORGET COLD FENCE

FSEEK (file offset mode -- prev-position | -1)

FSEEK is used to move the current location in a file at which subsequent calls to FREAD and FWRITE will operate.

The mode parameter, defined with the same names as referenced in the Amiga technical literature, is one of three types:

```
mode                interpret positional value as
```

```
OFFSET_BEGINNING   offset from beginning of file
OFFSET_END         offset from end of file
OFFSET_CURRENT     offset from current position
```

See chapter on File I/O for details.

```
( move 5 bytes forward )
MYFILE @  5 OFFSET_CURRENT   FSEEK .
```

Related Words: FREAD FWRITE

FSEEK? (file offset mode -- prev-position)

FSEEK? is functionally identical with FSEEK with one exception; if an error occurs during the seek operation, FSEEK? will print "Error on file seek!" and execute QUIT.

JForth provides for optional automatic cleanup of opened files if an error forces a QUIT. See MARKFCLOSE and UNMARKFCLOSE.

For further information, see FSEEK.

FWRITE (file addr #bytes -- #bytes-written | -1)

FWRITE will write #bytes from reladdr to the file specified by file-pointer at its current location.

FWRITE always returns a value. If the call was successful, it returns the number of bytes that were written; if an error occurred, it returns -1. An error is also indicated by a non-zero value in the user-variable FERROR, immediately following the call.

See chapter on File I/O.

Related Words: FOPEN 0FOPEN FSEEK FWRITE?

GETMODULE (-- , <modulename>)

See the chapter on MODULES.

GLOBAL-DEFER (<name> --)

Just like DEFER except DEFER stores its vector in the user area while GLOBAL-DEFER stores its vector in the dictionary. This distinction is important in multi-tasking Forths.

See section on DEFER.

Related Words: WHAT'S IS DEFER

GRAPHICS? (--)

```
GRAPHICS_LIB
GRAPHICS_NAME
```

Used to manage the Amiga Graphics Library . See LIBRARY .

HERE (-- addr)

Places the contents of DP (pointer to the next available dictionary location) onto the parameter stack. Indicates size of COMPILED image from the beginning of the kernel. The next word that is compiled will go "here".

```
CREATE FOO HERE  ( start simple array )
23 , 987 , 1722 ,  ( lay in some data )
HERE SWAP - CELL/ CONSTANT NUM_FOO
( figure out how many cells are in FOO )
```

HEX (--)

Sets user BASE to 16 decimal . All number input and output conversions will be done in hex .

```
DECIMAL 10 HEX . ( will display  A  )
```

Related Words: DECIMAL BINARY COMMAS BASE $

HICASE (-- n) "hi case"

HICASE is a CONSTANT used to determine the desired characteristics of output ASCII case. See OUTPUT-CASE.

HIDEMODULE (-- , <modulename>)

See the chapter on MODULES.

HOLD (char --)

Add the ASCII character to the number being converted to a text string. Used by # to store ASCII characters in memory.

```
: .TIME ( time -- )
    S->D <# # # ASCII : HOLD #S #> TYPE ;
1032 .TIME  ( prints 10:32 )
```

Related Words: <# # #S #> COMMAS

I (-- index)

Place current DO LOOP index on stack .

NOTE: this is the only approved method of retrieving the index value of the currently running DO-LOOP.

```
: PRINT-0-TO-9  ( -- )  10 0 DO  I .  LOOP CR ;
```

Related Words: J IK DO LOOP +LOOP -DO -LOOP

ICON?

```
ICON_LIB
ICON_NAME
```

Manage the Amiga Icon Library. See :Library

ID. (nfa --) "i d dot"

Print the name of the Forth word whose NFA is on the stack. COUNT TYPE won't work because of the smudge bits, etc.

Standards: fig '79 '83. Called .NAME or .ID in some '83 systems.

```
' NEGATE >NAME ID.
```

IF (flag --)

Execute the following code if the flag is true (non-zero). If the flag is false, then skip forward to the next ELSE or THEN. IF may be used during COMPILE MODE only, and will generate an error if used otherwise.

```
: TEST  ( n -- , print based on N )
    DUP 10 >
    IF . ."  is bigger than 10!" CR
    ELSE . ." is small." CR
    THEN
;
```

IF is an IMMEDIATE word and will execute immediately in COMPILE MODE. It has no effect on the data stack, but will store data needed to resolve the branch on a special stack called the "User Stack".

Compile time: user stack: (-- if-flag address)

Related Words: ELSE THEN = TRUE BEGIN ?PAIRS

IF-NOT (flag --)

Operates identically to IF , except the boolean flag satisfying the run-time branch is reversed.

Standards: JForth unique. Simply a faster and smaller NOT IF .

IF>ABS (relative-addr -- absolute-addr) "if to abs"

Convert address UNLESS it is NULL, ie. zero. This is typically used with Amiga routines that use an address or a NULL. The NULL has a special meaning and should not be converted.

Related Words: >ABS >REL IF>REL

IF>REL (absolute-addr -- relative-addr) "if to abs"

Convert address UNLESS it is NULL, ie. zero. NULL is often used as a flag so it must be preserved, not converted.

Related Words: >ABS >REL IF>ABS

IFLEAVELONG (-- var-addr)

See the chapter on CLONE.

IK (-- 3rd-loop-out-index)

Retrieve the index of the loop in which it is nested 3 deep. If I is for the current loop and J is for the next outer loop, then IK is for the loop next further out.

Standards: JForth unique

```
: EXAMPLE  ( -- )
    3 0
    DO   4 0
       DO  5 0
          DO  IK . J . I . CR
          LOOP
       LOOP
    LOOP ;
```

In the above example, the leftmost number will change the most slowly because it is the index of the outer loop.

IMMEDIATE (--)

Changes the word just compiled to be immediate. The programmer, by making a definition IMMEDIATE, sets the word to be executed, rather than compiled, when found within a colon definition. This is useful for extending the compiler.

```
: PRINT-HERE  ( -- , prints current DP, even while compiling )
    HERE .
; IMMEDIATE
: FOO 23 DUP PRINT-HERE + ;  ( will print HERE now )
```

To cause an IMMEDIATE word to be compiled, it should be preceded with [COMPILE] .

```
: FANCY-PRINT-HERE  ( -- , fancier version )
    ." Here = " [COMPILE] PRINT-HERE
;
```

IMMEDIATE? (nfa -- flag)

Given the name-field-address of a dictionary header, return whether it is compiled as an IMMEDIATE definition.

```
' LITERAL >NAME IMMEDIATE?  ( -- true  )  .  64  ok
' DROP    >NAME IMMEDIATE?  ( -- false )  .  0  ok
```

INCLUDE (<filename> --)

Input Forth code from the given file. This is used to compile source code from a file. If an INCLUDE command is encountered in a file it will include that file then continue with the original file. If you want to INCLUDE a file that has spaces in the name, then put the filename in double quotes.

```
INCLUDE RAM:X  ( compile what's in RAM:X )
INCLUDE "RAM:SPACEY FILE"
```

INCLUDE? (<name> <filename> --) "include question"

INCLUDE the given file if the Forth word named is not already compiled. This is useful if you need a word from a file, and want to compile it if it's not already loaded.

```
INCLUDE? MSEC JU:MSEC
INCLUDE? GR.INIT JU:AMIGA_GRAPH
```

If you are using ANEW in a file, you should place the INCLUDE? commands before you call ANEW .

Related Words: EXISTS? FILEWORD

INIT-USP (--) (? --US--) "init u s p"

Initialize the user stack. JForth incorporates a third stack mainly for use in compiling DO LOOP and other control structures.

Standards: JForth unique.

Related Words: US@ >US US>

INITCLONE (--)

See the chapter on CLONE.

INITIALIMAGESIZE (-- var-addr)

See the chapter on CLONE.

INLINE (--)

INLINE is an immediate compiler directive used to inform the compiler that the user wishes this word to always be compiled inline; i.e. the entire body of the definition (minus any trailing RTS) will be copied to HERE when later compiled. This avoids the overhead of a subroutine call and is thus faster.

INLINE may only precede the ; marking the end of a colon definition, and will cause the compiler to verify that the word may safely be compiled in this manner.

If so, the word will be marked INLINE by setting bit 31 in the size-field. Once marked, the word will unconditionally be compiled inline when referenced in later colon definitions.

If not, a warning message is issued, informing the operator that the word cannot be compiled inline, and the word will be set to CALLED .

Standards: JForth unique.

```
: EXAMPLE  ( -- )  ROT OVER +    INLINE ;
: FOO 23 34 45 EXAMPLE * . ;
( EXAMPLE will be expanded inline in FOO, not called. )
```

Related Words: ; BOTH

INLINE+ (n --) (addr -- addr+n)

INLINE+ adds the top of the data stack to the value on return stack. INLINE+ is one of a set of 4 words used to improve the readability and transportability of programs that use inline data.

INLINE@ (-- inline-data-address) "inline fetch"

INLINE@ returns the address of the inline data.

INLINE+ INLINE> >INLINE (INLINE is not related)

INLINE> (absolute-address --R--)

```
( -- inline-data-address )
```

INLINE> takes the return stack absolute address, converts it to a relative address, then moves it to the data stack.

Related Words: INLINE+ INLINE@ >INLINE (INLINE is not related)

INTERPRET (--)

This is a DEFERed word that is used to process a line of Forth input. See (INTERPRET).

Related Words: $INTERPRET QUIT COMPILING? INTERPRET? QUERY (INTERPRET)

INTERPRETING? (-- flag)

Returns true if STATE is zero, ie. interpreting, false if STATE is true, ie. compiling. When you are inside a colon definition, STATE is true. Usually used in IMMEDIATE words that behave differently when interpreting or compiling.

Related Words: STATE ?COMP

INTUITION? intuition question

```
INTUITION_LIB   intuition lib
INTUITION_NAME  intuition name
```

Used for managing the INTUITION Library. See :LIBRARY.

IS (cfa <name> --) "is"

IS provides the mechanism for setting the action vector for a DEFERed word. See DEFER.

```
' DROP IS EMIT
( sets the deferred word EMIT to execute DROP )
```

Related Words: DEFER CFA,

J (-- index)

Used in nested DO LOOPs. Gets the index of the DO LOOP one level above.

```
: TESTJ   2 0 DO  3 0 DO I . J . LOOP LOOP ;
TESTJ  ( Prints    0 0 1 0 2 0  <CR> 0 1 1 1 2 1 )
                   I J I J I J       I J I J I J
```

Related Words: I IK DO LOOP

KEY (-- char)

KEY is used by programs to obtain single-character input; this is taken from the AmigaDOS file-pointer stored in the variable CONSOLEIN, waiting until a key has been pressed.

KEY is a DEFERred execution word and is normally set to (KEY). Any word that replaces (KEY) should call FLUSHEMIT to force out any stored up characters.

```
: PAUSE ." Hit key to continue: " KEY DROP ;
```

If you clone a program with KEY, you may want to open a RAW: window to get immediate response. See CONSOLE! .

Related Words: ?TERMINAL EXPECT EMIT (KEY)

LATEST (-- nfa)

Returns the name-field of the most recently created dictionary header in the CURRENT vocabulary.

```
: FOO ;    LATEST ID.  ( Prints "FOO" )
```

Related Words: CURRENT HERE

LAYERS?

```
LAYERS_LIB
LAYERS_NAME
```

Standards: JForth internal

Used to manage the LAYERS library. See :LIBRARY .

LEAVE (--)

LEAVE a DO LOOP. Used in the form : ... DO ... LEAVE ... LOOP ; At run time LEAVE causes immediate exit from the loop with which LEAVE is associated .

```
: SCAN-MAX  ( max  -- , scan until found or max reached. )
    0 DO
        IS-FOUND?  ( -- true_if_found )
        IF ." Found at " I . CR LEAVE ( quit loop )
        THEN
    LOOP ;
```

Related Words: DO LOOP +LOOP RETURN

LIBVERSION (-- addr)

LIBVERSION is a user-variable, accessed by the XXX? functions (where XXX is the name of an existing LIBRARY, GRAPHICS? for example).

The JForth Library Manager normally opens the latest version of an existing library, but will settle for any version number.

To specify a particular version, the programmer should place the desired version number in LIBVERSION, just prior to calling the appropriate XXX? word.

Standards: Amiga unique

Related Words: :LIBRARY DOS

LIMIT (-- addr)

LIMIT is applicable only to the BLOCK or SCREEN environment, and returns the address of the end of the virtual buffer area.

```
PREV @ LIMIT =   \ check to see if time to 'circle' around
```

Related Words: FIRST BLOCK BUFFER (LIMIT)

LINESFILLV "lines fill virtual"

(file-pointer memory-block max-char-count -- #chars-actually-read)

This word is used to fill a memory area with ASCII text from a specified file. The file should contain standard EOL characters; the area is filled to a line boundary. The memory block MUST have been previously allocated via ALLOCBLOCK.

LINESFILLV accepts 3 parameters:

1) A file-pointer specifying the file to read.

2) The address of a memory-block to put the data read from the file.

3) The byte-count of the memory block area.

LINESFILLV is used by the READLINE function. See READLINE and the chapter on File I/O.

Standards: JForth unique

Related Words: OPENFV READLINE FWRITE FREAD FSEEK

LINK> (lfa -- cfa) "link to c f a"

LINK> converts the link-field-address of a word to the corresponding code-field-address.

```
' TASK  >LINK     ( derives the lfa of TASK )
( lfa -- ) LINK>  ( puts it back to the cfa of TASK )
```

Related Words: >LINK >NAME NAME> VLINK>'

LIT (n --inline--) (-- n) "lit"

LIT is compiled when the compiler must create code that serves to push a literal value to the stack at run-time. The value to be pushed is compiled in the dictionary after a call to LIT .

Standards: fig '79 83

Related Words: LITERAL INLINE@

LITERAL

Compile mode: (n --) Pops the value n from the stack, and compiles it such that it will be pushed at run-time.

Interpret mode: (n -- n) LITERAL has no effect when INTERPRETING.

LITERAL allows certain calculations to be performed once at compile time and stores only the result, along with a run-time operator, LIT, to push the result to the stack. This frees us from having to perform that same calculation repeatedly at run-time just to keep pushing the same value to the stack.

NOTE: LITERAL should not be used if the data to be compiled represents the address of a JForth entity. See the discussion of ALITERAL in the CLONE chapter.

Standards: fig '79 83

```
: TEN-DOZEN  ( -- ten-dozen )  [ 10 12 * ]  LITERAL   ;
```

Related Words: [ ] COMPILE IMMEDIATE LIT

LOAD (screen# --)

LOAD a screen of Forth code. LOAD is only applicable to the BLOCK or SCREEN environment; this word is not available in the standard JForth system...the source file JU:BLOCK must be compiled in ASCII-file format to gain access to LOAD.

LOAD causes interpretation to be redirected to the specified SCREEN of the file being pointed to by the user-variable SCR-FILE.

LOADing will continue to the next SCREEN of the file if the operator --> is encountered during interpretation of the current SCREEN.

Otherwize, LOAD will return when the end of the current SCREEN is reached, at which time interpretation will continue from wherever the LOAD command was invoked.

Standards: '83

Related Words: BLK >IN LIMIT BLOCK FIRST

LOCASE

LOCASE is a CONSTANT used to determine the desired characteristics of output ASCII case . See OUTPUT-CASE.

LONGCFA, (cfa --) "long c f a comma"

LONGCFA, is a DEFERred word; it normally executes (LONGCFA,) which acts to compile a long absolute JUMP-SUBROUTINE to the cfa on the stack.

Of the 3 types of calls available to the JForth compiler, this is the least preferred as it is slightly larger and slower and requires relocation.

For these reasons, LONGCFA, is only invoked by the compiler when the distance from the destination address is too great for the preferred methods to be practical.

Standards: JForth internal

```
\ works, but wastes time and space.
: HARD-VLIST ( -- ) [ ' VLIST LONGCFA, ] ;
```

Related Words: CFA, (LONGCFA,)

LOOP (--)

This is used with DO to form a loop that executes a specified number of times. Each time through the LOOP the index is incremented until it reaches the limit, at which point execution proceeds past the LOOP command. Use +LOOP if you want to add more than 1 to the loop index. See DO .

```
: HI ( -- , print HIYA 10 times)
    10 0 DO
        CR ." HIYA "
    LOOP ;
```

Related Words: DO +LOOP -LOOP

LPLACE (string-addr string-cnt destination-addr --)

LPLACE is functionally identical to PLACE, except it will unconditionally preserve ASCII-case across the move. See PLACE.

LWORD (char -- addr)

LWORD is functionally identical to WORD, except it will unconditionally preserve ASCII-case across the move. See WORD.

! " # $ % & ` ( ) * + ' - . / 0-9 : ; < = > ? @ AZ [\ ] ^ _ az { | } ~

M* (n1 n2 -- d1) "m times"

Multiply two signed single precision numbers N1 and N2 together and leave the signed double product D1.

M/ (d n1 -- rem n2) "m slash"

M/ divides a signed divisor N1 into a signed double dividend D to yield a signed remainder REM and the signed quotient N2.

M/MOD (d1 n1 -- rem d2) "m slash mod"

Divides a signed value N1 into a signed double-value D1 and returns the signed remainder REM and a signed double quotient D2 .

MAKEMODULE (size -- , <modulename>)

See the chapter on MODULES.

MAKEUCASE (-- addr)

This is a user-variable which controls whether WORD will convert all alphabetic characters to upper-case.

Normally, all dictionary headers are created by WORD. FIND for reasons of speed is CASE-SENSITIVE. Therefore, all words, by convention, are converted to upper-case as they are moved to HERE, by WORD.

The programmer, should he/she wish to receive lower-case strings at HERE, may either set (and restore when done) the contents of MAKEUCASE himself, or use the supplied version LWORD, which does this automatically.

```
MAKEUCASE @   FALSE MAKEUCASE !
PROGRAMMERS-GET-WORD
MAKEUCASE !
```

Related Words: WORD LWORD HERE

MAP (--)

MAP displays general information concerning current memory allocation and dictionary utilization, number of vocabularies, relocation information, file usage, the setting of the current MAX-INLINE variable, the state of the modules, and whether the HASHed vocabulary search is in use..

MARKFCLOSE (file-pointer --) "mark f close"

MARKFCLOSE places the FILE-POINTER in a list of file-pointers that will be closed if the system executes QUIT .

This mechanism allows an application to automatically clean up its resources in both of the two possible paths most programs may take... normal completion or an irrecoverable error abort.

Note that if the application terminates normally, it should clear this AUTO-CLOSE status for the file by executing UNMARKFCLOSE for each file it had previously marked.

```
FOPEN ?DUP       \ get the filename, did it open?
```

```
IF  DUP MYFILE ! \ yes, save pointer in my variable
    MARKFCLOSE     \ set it to auto-close at quit
    RUN-MY-PROGRAM \ do whatever I want to do with the file
    MYFILE @ DUP   \ need to do TWO thing to the pointer...
    UNMARKFCLOSE   \ remove it from the QUIT list...
    FCLOSE         \ and close the file
ELSE ." Can't open the file!"
THEN
```

Related Words: UNMARKFCLOSE

MARKFREEBLOCK (memory-block --)

MARKFREEBLOCK places the memory-block on a list of memory-blocks that will be closed if the system executes QUIT .

This mechanism allows an application to automatically 'clean-up' its resources in both of the two possible paths most programs may take... normal completion or an irrecoverable error abort.

Note that if the application terminates normally, it should clear this AUTO-FREE status for the memory-block by executing UNMARKFREEBLOCK for each memory-block it had previously marked.

```
0 1024 ALLOCBLOCK ?DUP  \ allocate 1K, did it allocate?
IF DUP MYMEM !     \ yes, save pointer in my variable
    MARKFREEBLOCK   \ set it to auto-free at quit
    RUN-MY-PROGRAM  \ do whatever I want to do with the mem
    MYMEM @ DUP     \ need to do TWO thing to the pointer...
    UNMARKFREEBLOCK \ remove it from the QUIT list...
    FREEBLOCK       \ and free the memory
ELSE ." can't allocate memory "
THEN
```

Related Words: UNMARKFREEBLOCK

MATCH? (addr1 count1 addr2 count2 -- addr | FALSE)

Search for string2 within a larger string1, leaving the address within string1 that matches if found, or FALSE if not found.

By setting contents of the USER variable MCASE-SENSITIVE true, a programmer can make the search be case-sensitive.

```
: LOOKFOR ( $string -- addr )
    COUNT  ( get address and count of big string )
    " RAIN" COUNT MATCH? ;
" THE RAIN IN SPAIN" LOOKFOR
```

Related Words: TEXT=? $= MCASE-SENSITIVE COMPARE

MATHFFP?

```
MATHFFP_LIB
MATHFFP_NAME
```

Standards: JForth unique.

These are used to manage to MATHFFP library.

MATHIEEEDOUBBAS?

```
MATHIEEEDOUBBAS_LIB
MATHIEEEDOUBBAS_NAME
```

Standards: JForth unique.

These are used to manage the MATHIEEEDOUBBAS library. See :LIBRARY.

MATHIEEESINGBAS?

```
MATHIEEESINGBAS_LIB
MATHIEEESINGBAS_NAME
```

Standards: JForth unique

These are used to manage the MATHIEEESINGBAS library. See :LIBRARY.

MATHTRANS?

```
MATHTRANS_LIB
MATHTRANS_NAME
```

Standards: JForth unique.

These are used to manage the MATHTRANS library. See :LIBRARY.

MAX (n1 n2 -- n1 | n2)

Compare the top two items on the stack, leave the largest signed value. You can use MIN and MAX to clip a value to a boundary.

```
23 7 MAX . ( print 23 )
( make sure x not negative )
CALC.X ( -- x ) 0 MAX SQRT
```

Related Words: MIN >

MAX-INLINE max inline

This is a USER-variable examined by the compiler when it is about to compile a reference to a word that may be either INLINE or CALLED. See (CFA,) . If a referenced word is defined as BOTH, and its size is over MAX-INLINE then a subroutine call to that word will be compiled. Otherwise it will be compiled as an inline macro. You can use this word to make trade offs between memory usage and speed. The higher MAX-INLINE is, the faster the code that is produced. Generally varying MAX-INLINE from 6-256 results in a size difference of 5-20% .

```
16 MAX-INLINE !
```

Related Words: (CFA,) BOTH INLINE

MAXVOCS max vocs

This is a system CONSTANT, equal to the maximum number of VOCABULARIES that may be defined.

Standards: JForth unique

Related Words: VOCABULARY

MCASE-SENSITIVE (-- addr)

This variable is used to control the case sensitivity of TEXT=?, a primitive for MATCH?.

See MATCH? TEXT=?.

MEASURE (<forth> --)

Measure the time it takes to execute the following code and print the time. This is handy for benchmarking code. Use BENCH if you are benchmarking things in a loop and need to subtract the overhead time.

```
\ Time how long it takes to compile it.
MEASURE INCLUDE JU:DEBUGGER
```

MEMF_CHIP

MEMF_CLEAR
MEMF_FAST
MEMF_PUBLIC

AMIGA Constants. See ALLOCBLOCK.

MENU.SETUP (0name menu# menu-addr --)

Initialize an Intuition Menu structure to reasonable defaults. See the chapter on Amiga Menus.

Related Words: EZMENU

MIN (n1 n2 -- n1 | n2)

Compares the top two stack values n1 and n2, leaving the smallest value.

```
( limit your deduction )
TAX-DEDUCTION @  MAX-ALLOWED @  MIN
```

Related Words: MAX

MOD (n1 n2 -- rem) "mod" or "modulus"

Leaves signed remainder REM on top of the stack after dividing N1 by N2.

```
13 5 MOD . ( print 3 )
```

Related Words:

```
/MOD */MOD M/MOD
```

MODE_NEWFILE MODE_OLDFILE

See FOPEN.

MOVE (addr1 addr2 u --)

Copies U bytes in memory starting at address ADDR1 to memory starting at address ADDR2.

MOVE will decide whether to copy from top to bottom, or vice versa, if the destination overlaps the source area. This will avoid the problem of data overwriting itself which can happen with CMOVE.

MOVE is also an optimized data transfer; under the right conditions, it will move data in excess of 1 megabyte/second on a standard Amiga 500.

Related Words: CMOVE CMOVE> WMOVE

MSEC (#milliseconds --)

Delay for approximately that many milliseconds. Delays over 20 milliseconds use the Amiga Delay() function. Delays less than 20 milliseconds have to use software timing and can err in accuracy. Do not use this where extreme accuracy is required. From file JU:MSEC.

```
1000 MSEC  ( wait one second )
```

MYDSP my d s p

This USER-variable is not implemented in JForth V2.0, but is reserved for future use.

MYRP my r p

This USER-variable is not implemented in JForth V2.0, but is reserved for future use.

MYTOS (-- addr) "my tos"

This USER-variable is not implemented in JForth V2.0, but is reserved for future use.

N>LINK (nfa -- lfa) "n to link"

Used to derive the link-field-address of a dictionary header from the name-field-address.

Related Words: LINK> >LINK ' >NAME

N>TEXT (n -- addr count) "n to text"

Convert a number to it's text equivalent in the current base. Like . except it doesn't type out the characters. This is useful when you want to write formatted files or draw numbers graphically.

```
GR.INIT GR.OPENTEST 20 30 GR.MOVE
1234 N>TEXT GR.TYPE  ( draw number )
```

Related Words: # . #DIGITS BASE EMIT LOGTO

NAME> (nfa -- cfa) "name to"

NAME> converts the name-field-address of a dictionary header to the code-field-address. The code-field-address is that place in a standard dictionary entry that contains the executable code for that definition.

```
." The most recently compiled code is "
LATEST DUP ID. NAME> DISM
```

Related Words: >NAME '

NEGATE (n -- -n)

NEGATE reverses the sign of N.

```
2 NEGATE .  ( prints -2 )
```

Related Words: DNEGATE ABS XOR

NEW

(--)

This word places MODE_NEWFILE in the user-variable FILEMODE. When the next file is opened using FOPEN, it will now create a NEW file instead of opening an existing one. See the chapter on File I/O.

CAUTION: do not set the NEW mode, unless a call to FOPEN, $FOPEN, or 0FOPEN immediately follows. Forgetting that NEW has been executed can result in the next file to be opened being inadvertently lost. The word OLD reverts filemode to MODE_OLDFILE, as do both FOPEN operations.

```
NEW FOPEN EMPTY-FILE
```

Related Words: FOPEN $FOPEN OLD

NEXTTASK

This USER-variable is not implemented in JForth V2.0, but is reserved for future use.

NFA (cfa -- nfa)

NFA has been renamed to >NAME to comply with the '83 standard. If you want NFA load JU:MULTISTANDARD.

NIP (a b -- b)

Remove second item from stack. Equivalent to, but faster than, SWAP DROP.

Related Words: SWAP DROP

NO@ (--)

Set the mode for local variable to not automatically fetch their values. They will leave their address instead. See the section on Local Variables.

Related Words: { } YES@

NO-COMMAS (--)

Turn off the use of commas when formatting numbers for output, by storing a FALSE in the user-variable (COMMAS).

Related Words: COMMAS (COMMAS)

NOCASE (-- n)

NOCASE is a CONSTANT used to determine the desired characteristics of output ASCII case . See OUTPUT-CASE.

NOCONSOLE (-- var-addr)

See the chapter on CLONE.

NOOP (--) "no op"

This is an empty compiled definition, generally useful for disabling deferred words that have no effect on the stack.

```
DEFER MY-VECTOR    ' NOOP IS MY-VECTOR
```

NOT (flag -- logically-opposite-flag)

Reverse the logic of the flag on the stack. This word acts to change a non-zero value to zero, and a zero to -1. This is a synonym for 0= .

NOTE: this is a departure from the '83 standard, which is to perform a 1's-complement operation on the flag. Load JU:MULTISTANDARD if you need the standard NOT .

```
BEGIN  MONEY-GONE? NOT
WHILE  SPEND-MORE
REPEAT
```

Related Words: 0= COMP

NUMBER ($addr -- d)

Converts a character string starting at address $ADDR into a double cell signed number using the value in BASE. The ADDR parameter points to a length byte. NUMBER is deferred and is changed when the floating point system is active.

If a decimal point is present in the string, the value in DPL will reflect the number of digits following it; otherwise DPL contains -1.

If the string cannot be converted, it is printed on the standard output device, and the ERROR exit is taken.

```
: ADDONE ( -- , convert add & print )
    BL WORD NUMBER  ( -- d )
    DROP 1+ . ;
ADDONE 1234  ( prints 1235 )
```

Related Words: ERROR NUMBER? CONVERT DPL

NUMBER? ($addr -- d true | false)

Attempt to convert the string at address $ADDR to a number, using the current value of BASE. If successful , return the double number and true; otherwise return false.

Standards: JForth unique.

Related Words: NUMBER CONVERT

OB.ARRAY (<name> --)

Create an instance, or object, of the class OB.ARRAY. This is a powerful type of array data structure. See the chapter on ODE.

OB.ELMNTS (<name> --)

Create an instance, or object, of the class OB.ELMNTS. This is a powerful type of multidimensional array data structure. See the chapter on ODE.

OB.STATS? (<member> -- offset #bytes)

Returns information about a structure member. This information is used by the compiler to compile proper references to Amiga 'C' structures.

```
GETMODULE INCLUDES
OB.STATS? NW_WIDTH .S
```

OBJECT (<name> --)

Create a basic OBJECT. OBJECT is the root class from which all other ODE objects are derived. See the chapter on ODE.

ODE

ODE is not an executable Forth word. It stands for the Object Oriented Development System. See the chapter on ODE.

ODD! (n addr --) "odd store"

Store a cell N at an odd or even address ADDR . Uses sequential BYTE accesses.

Normally not needed, since all JForth cell addresses are word aligned, but must be used if your code has a possibility of doing LONG references to odd addresses. The 68000 will generate an address error if you access word or long data at an odd address.

Standards: JForth unique. 68000 unique. 68020 will be able to use @ . Existence and definition depend on the host CPU type.

Related Words: @ ! D! D@ EVEN-UP ALIGN

ODD@ (addr -- n) "odd fetch"

Fetch a cell N from an odd or even address ADDR . See comments under ODD! above.

```
71 ODD@ . ( won't crash )
```

Related Words: ODDW@ ODDW! ODDD@ @ ! D! D@ EVEN-UP ALIGN

ODDD! (d addr --) "odd d store"

Store a double number at an odd or even address. See comments under ODD! above.

ODDD@ (addr -- d) "odd d fetch"

Odd addressable D@ . See comments under ODD! above.

ODDW! (n addr --)

Odd addressable W! See comments under ODD! above.

ODDW@ (addr -- n)

Odd addressable W@. See comments under ODD! above.

OF (value n --)

Used in conditional statement with CASE . See CASE .

OFF (var-address --)

Set the contents of the given address off.

```
FILEHEADERS OFF
```

Related Words: ON !

OFFSET_BEGINNING

```
OFFSET_CURRENT
OFFSET_END
```

(-- n1)

These are AMIGA constants, used to specify the type of FSEEK operation to perform. See FSEEK.

OLD (--)

This word places MODE_OLDFILE in the user-variable FILEMODE. This will cause FOPEN to attempt to open an existing file. See the chapter on File I/O.

```
OLD FOPEN AN-EXISTING-FILE
```

Related Words: FOPEN 0FOPEN NEW

ON (var-address --)

Set the contents of the given address to TRUE (-1).

```
VARIABLE DO-WINDOWS
DO-WINDOWS ON  ( set TRUE )
```

Related Words: OFF !

ONLY (--)

Remove all vocabularies from the context stack except ROOT. This is used to reset the vocabulary search path. See the section on Forth Vocabularies.

```
ONLY FORTH  ( set back to default )
```

Related Words: ALSO PREVIOUS FORTH ORDER

OPENFV (var-addr -- memory-block)

OPENFV is used to allocate a 1024-byte memory area that can later be used for a sequential virtual-buffer for fast file I/O. See the chapter on File I/O.

OR (n1 n2 -- n3)

Logical bit by bit or'ing of n1 and n2 . Result n3 is on top of stack . If a bit is on in either N1 OR N2 then the same bit will be on in N3.

```
BINARY 0011 1001 OR .    ( print 1011 )
( do something if key hit or input out of range )
100 >  ?TERMINAL  OR IF
```

Related Words: NOT AND XOR IF

ORDER (--)

Display the order of vocabularies that will be searched by the Forth compiler. It also shows the CONTEXT vocabulary which is where newly defined words will go. See the section on Forth Vocabularies.

OUT (-- addr)

User variable that contains a value incremented by EMIT, OUT indicates the column number that the cursor is currently in. OUT is reset by CR .

```
: GOTAB  ( -- , move to next tab )
    OUT @ 8 MOD
    8 SWAP - SPACES ;
```

Related Words: EMIT CR CR? >NEWLINE +OUT

OUTBUF (-- addr)

Returns the address of the output buffer being used by the FAST I/O mode. See FAST.

OUTPUT-CASE (-- var-addr)

This is a USER-variable that may be used to force characters that are EMITed to one of the desired output forms:

1) NOCASE - output in mixed case, no conversion (default)

2) LOCASE - output appears in lower case only.

3) HICASE - output appears in upper case only.

```
HICASE OUTPUT-CASE !
." Hello Fred" CR
```

OVER (a b -- a b a)

Copy second item on stack to top. Leapfrog.

```
11 22 OVER .S  ( print 11 22 11 )
```

Related Words: DUP ROT

PAD (-- addr)

PAD returns the address of a temporary work area. PAD is defined as HERE + 128. Be careful HERE doesn't move when you are using PAD because it will move too. Immediate strings are stored at PAD. Numeric strings are built just below PAD. PAD extends up until it runs into the data stack area.

```
" Hello" . PAD .
```

Related Words: #K HERE SP@

PARSE (char -- addr count)

PARSE uses the value of >IN as an index into the address and count returned by SOURCE, parsing from that location until either CHAR is found or the string is exhausted.

PARSE returns the address ADDR of where it started looking and the COUNT of the characters until CHAR was found.

PARSE does not skip over leading occurrences of CHAR. It returns even if the first character checked is CHAR .

```
: PS" ( -- , like ." )
   ASCII " PARSE TYPE ;
20 5 - PS" Answer = " . CR
( print   Answer = 15 )
```

Related Words: /STRING SOURCE SKIP SCAN WORD QUERY

PARSE-WORD (char -- addr count)

PARSE-WORD is just like PARSE except it will skip over skip over leading delimeters. IF the first character it finds equals char , it will continue checking until a non-delimiter is found. It will then again look for a delimiter, after which it will return.

Related Words: /STRING SOURCE

PICK (... v2 v1 v0 M -- ... v2 v1 v0 vM)

Create a copy of value number M, replacing M on the stack. Value number 0 is the value immediately below the PICK index.

!!! WARNING !!! - This '83 version of PICK differs from old standard versions of PICK. In old versions, the top value was value number 1 not 0 . Please see the MULTISTANDARDS facility for access to different versions of PICK . The following code shows the equivalent of doing a DUP.

```
OLD WAY     vs    NEW WAY
1 PICK            0 PICK
```

Standards: '79 '83

```
33 45 66 87   2 PICK .  ( print 45 )
```

Related Words: RPICK XPICK ROT

PLACE (source-address count $dest-addr --)

PLACE is used to move a string from one address to another. If the user-variable MAKEUCASE is non-zero, it will convert the string to upper-case after the move.

Instead of manipulating MAKEUCASE, the programmer may call LPLACE, which will not perform the conversion, allowing lower-case output.

Standards: JForth unique

```
CREATE SPAD 80 ALLOT   \ make string holder
" Hello World" COUNT SPAD PLACE
SPAD COUNT TYPE   \ print copied string
```

Related Words: $MOVE MOVE COUNT

PREVIOUS (--)

Drops the vocabulary added to the context stack by ALSO. This provides a way to unnest changes to the compiler search order. See the section on Forth Vocabularies.

```
ALSO MUSIC   \ search music vocab first
INCLUDE COMPOSITION
PREVIOUS     \ now back the way it was
```

POP (memblock -- n)

POP is used on a memory block that had been allocated via ALLOCBLOCK, and returns the cell just under that being pointed to by FREEBYTE, adjusting the FREEBYTE counter.

POP, in conjunction with PUSH, allows memory of this type to be conveniently used as stacks. Using the contents of the FREEBYTE field of a memblock, these words keep track of the next free location.

Note that, for efficiency , no error checking is performed by POP. It is the responsibility of the calling program to insure data is present before trying to POP it.

Standards: JForth unique.

```
\ Close a stack of files.
MY-FILES @   DUP FREECELL 0  ( -- memblock #cells 0 )
DO   DUP POP ( -- memblock file-pointer ) FCLOSE
LOOP DROP
```

Related Words: PUSH ALLOCBLOCK FREEBYTE +STACK -STACK

POTGO?

```
POTGO_LIB
POTGO_NAME
```

These words are used to manage the POTGO Library. See :LIBRARY.

PRINTER.ON

Echo JForth output to both the screen and printer. Uses $LOGTO to vector EMIT .

```
INCLUDE JU:LOGTO
PRINTER.ON   MAP   PRINTER.OFF  ( print MAP on printer )
```

If you want print a file on the printer, you can use the AmigaDOS copy command.

```
COPY  filename  PRT:
```

If you want to print a file with line numbers and proper page breaks, you can use the JForth Print Application. See the appendix on the Clonable Applications.

PULLTIB (--) "pull t i b"

PULLTIB restores the previous TIB environment that had been saved via PUSHTIB.

PULLTIB causes the TIB, #TIB, >IN and FBLK to be restored.

Standards: JForth unique

```
PUSHTIB   QUERY INTERPRET  PULLTIB
```

Related Words: PUSHTIB #TIB TIB

PUSH (n memblock --)

PUSH is used on a memory block that had been allocated via ALLOCBLOCK, and places N in the location being pointed to by the base address+FREEBYTE, adjusting FREEBYTE to the next available location.

PUSH, in conjunction with POP, allows memory of this type to be conveniently used as stacks.

Note that, for efficiency sake, no error checking is performed by PUSH. It is the responsibility of the calling program to insure that room exists in the allocated memory space before PUSHing to that area.

Standards: JForth unique.

```
( -- file-pointer )  MY-FILES @  DUP FREEBYTE OVER SIZEMEM <
IF   PUSH
ELSE ." FILE-STACK FULL!  NO ROOM FOR ANOTHER PUSH!" QUIT
THEN
```

Related Words: FREEBYTE FREECELL SIZEMEM POP ALLOCBLOCK

PUSHRELOC (rel-addr --) "push reloc"

This creates a relocation entry for an address that requires relocation by the Amiga Loader as JForth is being loaded in from mass storage.

Occasionally, in a large image, it is necessary to compile an absolute 32-bit address, referencing another part of JForth, within the body of a definition.

This potentially could pose a problem, as programs cannot specify their destination when being loaded, and any absolute references will be invalid.

JForth solves the problem by keeping a table of all addresses needing relocation. To add to this table, just pass to PUSHRELOC the JForth-relative address of the location that contains the 32-bit reference. Note that the location should be properly initialized with the correct 32-bit absolute address when PUSHRELOC is called.

Standards: Amiga unique

```
VARIABLE IMAGE-BASE   0 >ABS IMAGE-BASE !  IMAGE-BASE PUSHRELOC
```

PUSHTIB (--)

PUSHTIB saves information about the TIB and INTERPRET environment. Input can then be redirected to a new level, then each environment restored by a corresponding call to PULLTIB.

This technique is used in such nestable system words as INCLUDE and ?PAUSE.

PUSHTIB is nestable to a level of 16.

Standards: JForth unique

```
PUSHTIB   QUERY INTERPRET  PULLTIB
```

Related Words: PULLTIB #TIB

QUERY (--)

QUERY will repeatedly receive characters from the keyboard, placing them in the TIB until a carriage return is entered. Upon exiting, #TIB will contain the number of characters received, and >IN will be 0.

Note that calling QUERY while a valid INTERPRET stream exists in the TIB will result in the loss of the remainder of the stream, unless PUSHTIB and PULLTIB are utilized before and after the QUERY operation and subsequent user processing.

```
: GET-NUMBER ( -- d ) ." Input a number: "
   QUERY  BL WORD NUMBER ;
```

Related Words: EXPECT TIB >IN PUSHTIB PULLTIB #TIB

QUIT (-- , quit whatever and return to interpreter)

QUIT is a DEFERred word, whose default content in JForth is (QUIT). When a program has sensed an irrecoverable error, QUIT is the function used to terminate program flow, and restart the system from a known point. In a cloned application, QUIT will cause an exit from the program. For more information specific to QUIT in the development environment, see (QUIT).

```
0< IF ." Error in this word!" QUIT THEN
```

Related Words: INTERPRET ABORT CLONE

R (-- n) (n --R-- n)

Copies the top of the return stack onto the top of the data stack. As this name has been superseded by R@, we do not recommend R.

Standards: FIG , R@ in '79 and '83.

Related Words: >R R> R@

R# (-- addr) "r sharp"

R# is a USER-variable that is only applicable in the BLOCK environment.

After the source file SCRED has been loaded, it is used to store the current cursor position (relative to the start of the BLOCK) in the supplied, FIG-like line editor.

```
R# @ C/L /  ( leaves line# cursor is on )
```

Related Words: LOAD BLOCK Used by many words in line editor and SCRED.

R> (-- n) (n --R--) "r from"

Takes a value N off of the return stack and pushes it onto the data stack. WARNING Only remove thing from the return stack that you put there with >R . Otherwise the subroutine return mechanism will fail. The return stack is a handy place to temporarily store values.

```
: UNDER1+  ( a b -- a+1 b )
    >r  ( save B ) 1+ r> ;
```

Related Words: >R R@ RDROP

R0 (-- addr) "r zero"

USER variable containing the initial value for the return stack pointer. This is used internally by JForth System Manager

Related Words: RP!

R@ (-- n) (n --R-- n) "r fetch"

Copy top of return stack onto top of stack .

Standards: '83 and '79 . FIG uses R

```
: EXAMPLE ( -- )  5 >R R@ ( will be 5 ) RDROP ;
```

Related Words: >R R> R

RANDOM (-- rnd)

Generate a 16 bit pseudo-random number using the linear congruential method. This is in JU:RANDOM. The sequence is based on the variable RAND-SEED.

Related Words: CHOOSE

RANGEOF (value low high -- value |)

Executes the following code up to an ENDOF, and drops the value, if the value falls within the given range. The range includes LOW and HIGH. See CASE.

RAWEXPECTECHO (-- var-addr)

See the chapter on CLONE.

RDROP (--) (n --R--) "r drop"

Drop top item N off the return stack. Only drop things you put there or the subroutine mechanism will fail.

```
: EXAMPLE ( -- )  5 >R RDROP ;  ( won't crash )
```

Related Words: DROP XRDROP >R R> R@

READLINE "read line"

(file-pointer var-addr addr maxlen -- addr #read | addr -1 if EOF)

```
file-pointer  specifies the file to read.
var-addr  holds the address of the virtual-buffer.
  addr = the place in memory to place the data
maxlen = maximum # characters to read
```

READLINE is used to read the next line of a file into memory, stopping at the EOL character, or if the memory limit is reached. The memory is read through a 1K sequential virtual-buffer that has been allocated via OPENFV . See OPENFV .

The memory address is passed through. In addition, if not at end-of-file, the number of characters that were actually read is returned. Otherwise a -1 is returned. Note that an empty line will return 0, a valid line-length.

Virtual memory areas allocated for reading should be closed via CLOSEFVREAD .

See chapter on File I/O for more information.

```
: NEXTLINE   ( -- flag , read and display next line )
    MY-FILE @  MY-BUFFER  PAD  1000  READ-LINE
    ( -- addr n1 )   dup 0>
    IF TYPE TRUE
    ELSE 2DROP FALSE
    THEN ;
```

Related Words: OPENFV LINESFILLV CLOSEFVREAD FREAD

RECURSE (--)

Calls word currently being defined. Recursion is powerful tool but must be used with caution. The rules for recursion are:

1) Do NOT use global variables because they will be changed by recursive calls. Use Local Variables instead.

2) Provide a way to stop recursion. Excessively deep recursion will overflow the return stack.

Here is an example of using recursion to calculate N factorial. N Factorial is defined as:

```
N*(N-1)*(N-2)....*1
```

We can define it recursively as:

```
FACT(N) = N * FACT(N-1)
```

```
FACT(1) = 1
```

Here is the Forth code. If this is the first time you have used recursion, it will seem a little mind bending. LISP programmers do this stuff all the time.

```
: FACT  ( N -- N-factorial )
   dup 1 >
   IF
      dup 1- recurse *
   THEN
;
4 FACT . \ should print 24
```

RECURSIVE (--)

This is not in the dictionary because it is so easy to define. When a word is being defined, it is SMUDGED so that if the compilation fails it can't be called. By calling UNSMUDGE, we can allow a word to call itself. Compare this example to the example under RECURSE.

```
: RECURSIVE  ( -- ) unsmudge ; immediate

: FACT  ( N -- N-factorial )
   RECURSIVE  \ make this word recursive
   dup 1 >
   IF
      dup 1- FACT *  \ calls itself!
   THEN
;
```

REDEF? (-- addr)

This user-variable is scanned by (CREATE) . If a Forth word is redefined a message will be printed if this variable if TRUE.

```
REDEF? OFF
: FOO ;
: FOO ;   ( NO message!)
```

REPEAT (--)

Used to terminate a BEGIN ... WHILE ... REPEAT conditional construct. Unconditionally branches to code following BEGIN. See WHILE for example.

For advanced users: REPEAT is actually an IMMEDIATE word that has the following stack diagram.

(addr-begin begin-flag waddr while-flag --)

```
addr-begin = address following BEGIN to branch to
begin-flag = shows BEGIN started the loop
     waddr = address of 0BRANCH offset created by WHILE
while-flag = pairs checking signature for WHILE
```

Related Words: BEGIN WHILE UNTIL IF ELSE THEN DO LOOP

RETURN (--)

RETURN causes an immediate exit from the the current colon definition, even from within nested DO LOOP's . Like the other control structures, IF ELSE LOOP etc... , It has an immediate part and a run time part. RETURN figures out the LOOP nest level at compile time, then compiles the appropriate number of return stack items to drop. If used outside any LOOP structures, RETURN compiles EXIT . The run time action is to exit the function when executed.

```
Run time:
  Stack: ( -- )
  Return stack: ( return-address loop-parameters... --R--)
Compile time
  Stack: ( -- )
  Return stack: ( --R-- )

      : EXAMPLE   ( -- )  10 0
          DO  100 0
              DO  I . ." about to return, bye" RETURN
              ." won't get here"
              LOOP ." won't ever print this"
          LOOP  ." Also won't print this" ;
          ( different from LEAVE)
```

Related Words: ?RETURN EXIT ?EXIT LEAVE

REVERTVOC (-- addr)

REVERTVOC is a USER-variable that is examined by : "colon". If found true, it will, at the very beginning of the definition, change the vocabulary that will be searched to the CURRENT vocabulary. The default state of REVERTVOC is FALSE. REVERTVOC has been provided for compatibility with older vocabulary systems.

Related Words: : VOCABULARY DEFINITIONS ORDER

ROOT (--)

ROOT is the root vocabulary of JForth. It will always be searched, even if the word ONLY was used. See the section on Vocabulary for more information.

Standard: '83 experimental proposal.

Related Words: VOCABULARY ORDER VOCS

ROT (a b c -- b c a)

Rotates the third item to the top of the stack.

```
11 22 33 ROT . ( prints 11 )
```

Related Words: SWAP DUP OVER PICK >R R>

RP! r p store

RP! has two different meanings, based on the standard you are using.

Standards: FIG and 79 (JForth default) ...

(--)

Initializes the return stack pointer to the address contained in R0 . Used in COLD and QUIT .

'83 (available from JU:MULTISTANDARD file)

(addr --)

Sets the return stack pointer to the addr on the data stack.

Related Words: R0 COLD QUIT

RP@ (-- addr) "r p fetch"

RP@ pushes the return stack pointer address on top of the stack .

RPICK (n -- rval-n)

This is the return stack equivalent of PICK. RPICK uses the value N on the stack to copy the contents of cell number N out from the return stack and push the value on top of the stack . This is the only way to access items deep down on the return stack . This is zero based so a 0 RPICK is equivalent to an R@ . RPICK is very fast and efficient, compared to using multiple calls to R> .

```
: EXAMPLE ( -- )
    5 >R 6 >R 7 >R 2 RPICK . ( 5 ) 3 XRDROP ;
```

S->D (n -- d) "s to d" "single to double"

S->D sign extends a 32 bit value N into a 64 bit double number D . This means the MSB of N is duplicated through the upper half of D .

```
-23 S->D D.  ( prints -23)
```

Related Words: W->S B->S

S0 (-- addr) "s zero"

S0 is a user variable containing the initial value of the stack pointer.

Related Words: SP@ 0SP DEPTH R0

SAVE-FORTH (<filename> --)

Save the current JForth dictionary in a file that can be executed later.

This yields two practical uses in the JForth development cycle:

Useful utilities and in general, any program may be saved in compiled form, and therefore almost instantly brought up . This will prove valuable, as you can save a compiled image with just the routines you've tailored for developing and debugging your application.

SAVE-FORTH is a vital part of the mechanism for expanding dictionary size. As your program grows in size, you may receive the "insufficient dictionary available" message. The following procedure lists the steps for increasing your JForth image size:

1) Set the #K variable...it tells SAVE-FORTH how large an image to save. In our example, we simply increment #K by 20 with +! ...

```
20 #K +!
```

Our new dictionary will be 20 X 1K (1K=1024) or 20,480 bytes larger. Note: you can also place the absolute value for size in #K ... if it contained 100 , we could type:

```
120 #K !
```

This would have the same result.

2) Save the image to the disk, using your own filename:

```
SAVE-FORTH MYFORTH
```

NOTE: if #K holds 120, you will need slightly more than 120K of disk area (about 5% more).

3) Exit JForth, returning to the CLI with:

```
BYE
```

4) From the CLI, run the image you just created with:

```
RUN MYFORTH
```

JForth will once again appear, (with the same words.) The only difference will be the larger dictionary area. Of course, you are using that much more Amiga memory now. Note: if AmigaDOS says that you don't have enough memory to run your new image, try rebooting the system...this recovers any fragmented memory.

Trivia: By setting #K to less than the current HERE (0 is a convenient number), SAVE-FORTH will create a minimum-sized image, occupying as little disk space as possible. This image however, will have an available dictionary size less than 2K when later booted. This could be useful for saving complete applications that you plan to use without adding much to their dictionary.

Standards: JForth unique

Related Words: #K #U MAP

SAVE-IMAGE (<wordname> <filename> [options] --)

Save a cloned program to a file. See the chapter on CLONE.

SCAN (addr count char -- addr' count')

```
addr = an address
count = length to search
char = an ASCII character
addr' = address where char was found OR end+1
count' = string length after matching char, 0 if not found
```

SCAN the string at ADDR (that is COUNT bytes long) for CHAR.

If found, return the address that matched and the REMAINING LENGTH of the string including the CHAR. If not found, return the end-of-string+1 for ADDR' and a COUNT' of 0 .

Standards: F83

```
" A text string" COUNT ASCII r  SCAN  TYPE
( The above would print "ring" )
```

Related Words: SKIP PARSE PARSE-WORD COMPARE QUERY

SCAN-ALL-VOCS (--)

SCAN-ALL-VOCS is a JForth Unique generalized function for accessing all the words in JForth. Two deferred words are used. WHEN-VOC-SCANNED is executed when each vocabulary is scanned, WHEN-SCANNED is executed when each word is scanned. By setting these words you can write systems that will process all of the words in the dictionary. Please see the chapter on Vocabularies for more info.

Related Words: WHEN-SCANNED WHEN-VOC-SCANNED SCAN-VOC SCAN-WORDS

SCAN-WORDS (--)

SCAN-WORDS is a JForth Unique generalized function for accessing the words in the CONTEXT vocabulary. The deferred word WHEN-SCANNED is executed when each word is scanned.

Related Words: SCAN-VOC SCAN-ALL-VOCS WORDS CONTEXT

SCAN-VOC (--)

SCAN-VOC is a JForth Unique word used by SCAN-ALL-VOCS and SCAN-WORDS to search a vocabulary. The deferred word WHEN-SCANNED is executed when each word is scanned.

Related Words: WHEN-SCANNED WHEN-VOC-SCANNED SCAN-VOC SCAN-WORDS

SEALMODULE (-- , <modulename>)

See the chapter on MODULES.

SET-BIT (n bit# -- n')

Set a specific bit in N. Bit 0 is the LSB.

```
0 4 SET-BIT . ( 16 )
```

Related Words: CLR-BIT @BITS !BITS BIT-SET? BIT-CLR? OR AND XOR

SHIFT (n shift-count -- n')

SHIFT n the number of bits indicated by shift-count, shifting in a zero as the new bit. If shift-count is positive, the shift is to the left, otherwise it is to the right.

Useful, for example, when extracting or inserting bit fields.

Standards: 83 suggested

```
HEX 75  DUP  0F AND .   ( print 5)
-4 SHIFT 0F AND .       ( print 7)
```

Related Words: ASHIFT 2* 2/

SHOWME (-- , <wordname>)

See the chapter on CLONE.

SIGN (sign dbl -- dbl)

If 'sign' is negative , add a minus sign to the output string. The double number 'dbl' is not affected. Used in numeric conversion.

Standard: FIG. In '83 (n --).

```
: N>TEXT  ( N -- addr count , signed number to text )
    S->D   SWAP   OVER  DABS
    <#  #S  SIGN  #> ;
-234 N>TEXT TYPE
```

Related Words: HOLD # #S

SIZEMEM (memory-block -- allocated-size)

Given the address of a memory-block that had been acquired via ALLOCBLOCK, return the size of the allocated block. See chapter on Memory Management.

```
MY-MEM @ SIZEMEM  ( -- size )
```

Related Words: ALLOCBLOCK FREEBYTE FREEBYTEA FREEBLOCK

SKIP (addr count char -- addr' count')

```
addr = an address
count = length to search
char = an ASCII character
addr' = address of 1st char that didn't match OR end+1
count' = string length after non-matching char,
       0 if entire string matches
```

Remove leading characters from a string.

Scan through the string at ADDR (that is COUNT bytes long) for CHAR, SKIPping leading occurrences until a non-matching character is found, or the end of the string. If a non-matching character is found, return its address and the REMAINING LENGTH of the string. If the entire string matches, return the end-of-string+1 for addr' and a count' of 0 .

Standards: F83

```
"     Finally a word!" COUNT BL SKIP  TYPE
( Don't type out leading spaces!)
```

Related Words: SCAN PARSE PARSE-WORD COMPARE FIND

SKIP-WORD

SKIP-WORD causes the next pass through WORD or FILEWORD to ignore the input stream, accepting instead the existing text at HERE.

```
: $CREATE ( $name -- , create a word whose name is on the stack
)
   HERE $MOVE
   SKIP-WORD CREATE
;
" MYDATA" $CREATE 234 ,
MYDATA @ .
```

Related Words: WORD LWORD FILEWORD UNWORD $FOPEN

SLOW (--)

SLOW reverts the EMIT I/O to a conventional single-character scheme. JForth default I/O technique is line-buffered, see FAST. In SLOW mode, emit will output each character as soon as it it is called. This might be handy in debugging, continuously updated displays, etc.

You may want to use FLUSHEMIT to force output in FAST mode instead of using SLOW.

Standard: JForth unique .

Related Words: (EMIT) (KEY) FAST EMIT TYPE

SMUDGE (--)

Mark the LATEST definition so that FIND will be unable to locate it. SMUDGE means to hide or conceal the latest definition . SMUDGE is used in : and CODE and some other defining words to make sure an incomplete definition is not accidentally compiled or executed.

Hashing may cause unpredictable results with SMUDGE , see HASH.OFF.

Standards: FIG and '79 use SMUDGE but they TOGGLE the smudge state, JForth sets it. '83 systems use HIDE .

```
VARIABLE BAD-MAKE
: RISKY-WORD  ( -- flag , true if fails )
    bad-make @ ;
: MAKEWORD
    CREATE  SMUDGE ( Make unFINDable )
        risky-word abort" Couldn't MAKEWORD!"
        UNSMUDGE   ( Make FINDable if survived RISKY-WORD)
    DOES> .
;
MAKEWORD OKWORD
OKWORD    ( Will be found!)
TRUE BAD-MAKE !  MAKEWORD BADWORD   ( Creation fails)
BADWORD   ( Won't be found!)
```

Related Words: UNSMUDGE REVEAL FIND CODE :

SOURCE (-- addr count)

This is a DEFERred word; by default it executes (SOURCE).

SOURCE is used by PARSE and PARSE-WORD to return the address and length of the available input stream. (SOURCE) normally returns the TIB start and the contents of #TIB .

Related Words: PARSE PARSE-WORD IS WHAT'S TIB #TIB QUERY WORD

SP! "s p store"

SP! has two different meanings, based on the standard you are using.

FIG and '79 (JForth default) ...

(--)

Initializes the data stack pointer to the address contained in S0 . Used in COLD and ERROR .

'83 (available from JU:MULTISTANDARDS file)

(addr --)

Sets the data stack pointer to the addr on the data stack. Use 0SP in place of the old style SP! .

Related Words: 0SP SP@ S0 COLD

SP@ (-- addr) "s p fetch"

In a traditional Forth this is the address of TOS , the Top Of Stack. In JForth, however, TOS is cached in register D7 inside the 68000. It has no "address". SP@ , then, actually returns the address of the SECOND item on the data stack in JForth.

Try to avoid using SP@ , use PICK and DEPTH if you can.

Related Words: RP@ DEPTH S0

SPACE (--)

Outputs an ASCII space, 20 hex, to the current EMIT device.

Related Words: SPACES EMIT BL ASCII

SPACES (n --)

Prints N spaces on the standard EMIT device . Ascii space = 20 hex .

Related Words: SPACE EMIT TYPE

SPAN (-- addr)

SPAN is a user-variable that is set to the number of characters last read in by EXPECT.

```
: GET$ ( -- $string , input string )
    PAD 1+ 120 EXPECT ( place characters at PAD+1)
    SPAN @ PAD C!     ( update byte count at PAD)
    PAD ;
```

Related Words: EXPECT QUERY

SPARE (-- addr)

Literally a spare USER variable for temporary use by any program .

```
FOPEN RAM:TEMP    SPARE !   ( handy place)
```

Related Words: USER PAD >R

SPEAK ($string --)

Translate string to phonemes and speak them. This uses the Amiga Narrator device. Make sure you have the volume up. Check out JD:DEMO_SPEAK for examples.

```
INCLUDE? SPEAK JU:SPEAK
SPEAK.INIT  ( -- , allocate I/O request block )
" Greeting Earthlings!" SPEAK  ( etc. )
" We come in pea  AAayaueahh!" SPEAK
SPEAK.TERM  ( deallocate when through
```

SQRT (N -- N1/2) "square root"**

Take the integer square root of N. Truncates answer. You can scale N by 10,000 to get two "decimal places" of accuracy.

```
INCLUDE? SQRT JU:SQRT
49 SQRT .   ( prints 7 )
```

STACKSIZE (-- var-addr)

See the chapter on CLONE.

STATE (-- addr)

A user variable used by INTERPRET, reflecting the state of the interpreter.

The JForth interpreter may be in one of two states;

1) If STATE = zero, the interpreter is in INTERPRET Mode, executing words as they are parsed from the input stream.

2) If STATE = non-zero, the interpreter is in COMPILE Mode, compiling words as they are parsed from the input stream.

Standards: '79, '83, In FIG 0 means system is compiling .

```
: NAMEOF ( <word> -- nfa )
    BL WORD FIND   ( search dictionary )
    IF  >NAME STATE @  ( compiling mode? )
        IF [COMPILE] LITERAL  ( save in dictionary )
        THEN
    ELSE COUNT TYPE ."  not found"
    THEN
; IMMEDIATE
NAMEOF SWAP ID.
: FOO NAMEOF SWAP ID. ; ( compile SWAP's NFA as LITERAL )
FOO       ( print "SWAP" )
```

Related Words: INTERPRETING? COMPILING? [ ]

STATS (--)

See the chapter on CLONE.

SWAP (a b -- b a)

Exchange top two stack items.

```
23 56 .S
SWAP .S
SWAP .S
```

TASK (--)

TASK is an empty definition, below which resides the JForth Kernal (that part of JForth which can not be VIEWed or re-generated by the programmer).

! " # $ % & ` ( ) * + ' - . / 0-9 : ; < = > ? @ AZ [\ ] ^ _ az { | } ~

The source code for the JForth system is provided for all words above TASK, and the system can be regenerated from this point from the JF: directory on the EXTRAS disk.

TEMPBUFF (-- addr)

TEMPBUFF is a user-variable, provided as a convenience for the programmer.

The programmer may allocate a virtual-buffer via OPENFV, and place the resultant address in TEMPBUFF .

Subsequently, the programmer may access file-virtual words that directly access TEMPBUFF, providing simple stack diagrams (they don't require the buffer or a variable address parameter). The only such word provided is TEMPF, ; the programmer may easily add more.

Note: TEMPBUFF usage is non-reentrant; you must save its' contents if you call another word that will use it.

Standard: JForth unique

Related Words: TEMPFILE TEMPF, OPENFV

TEMPF, (n --) "temp f comma"

This word writes the 32 bit value N to the file whose file-pointer is contained in TEMPFILE.

The write operation will be done through a virtual-buffer, opened with OPENFV, whose address is contained in TEMPBUFF.

NOTE: TEMPF, usage is non-reentrant. You must save the contents of TEMPFILE and TEMPBUFF if you call another word that will also need it.

Standard: JForth unique

Related Words: TEMPBUFF TEMPFILE OPENFV CLOSEFVWRITE

TEMPFILE (-- addr)

TEMPFILE is a user-variable, provided as a convenience for the programmer.

The programmer may open a file via FOPEN or (FOPEN), and place the resultant file-pointer in TEMPFILE.

Subsequently, the programmer may access file-operation words that directly access TEMPFILE, providing simple stack diagrams (they don't require the file-pointer parameter). The only such word provided is TEMPF, ; the programmer may easily add more.

NOTE: TEMPFILE usage is non-reentrant. You must save its contents if you call another word that will use it.

Standard: JForth unique

Related Words: TEMPBUFF TEMPF,

TEXT=? (addr1 count addr2 -- flag)

Compare two strings, each count bytes long, return a TRUE if they match, FALSE otherwise.

TEXT=? will perform a case-sensitive compare if the value of MCASE-SENSITIVE is non-zero; otherwise the compare will be performed ignoring ASCII case (JForth default condition).

```
: FROG=?  ( $word --flag )
   count 4 =
   IF " frog" count swap text=?
   ELSE drop false
   THEN
;
" bird" frog=? .
" Frog" frog=? .
```

Standard: JForth unique

Related Words: MATCH? MCASE-SENSITIVE $= COMPARE PARSE SKIP SCAN

THEN (--)

Mark the end of an IF...THEN or an IF...ELSE...THEN conditional construct. See tutorial for more examples.

```
: EXAMPLE  ( N -- )
    5 =
    IF  ." It's a five! "
    ELSE ." Why didn't you type in five? "
    THEN
    cr ." Thank you." ( executed in any case ) ;
```

Related Words: IF ELSE BEGIN UNTIL WHILE REPEAT

TIB (-- addr) "terminal input buffer"

TIB returns the memory area used for receiving and storing the input stream that will be INTERPRETed.

```
TIB 100 TYPE  ( type this line plus previous stuff)
```

Related Words: #TIB TIB0 SOURCE

TIMER?

```
TIMER_LIB
TIMER_NAME
```

Internal operators to manage the TIMER library. See :LIBRARY.

TIMES (n --)

Repeat the current command line N times. TIMES should be placed at the the end of the command line. TIMES can only be used from the keyboard.

If the command-line is to repetitively operate on a number or set of numbers, they must exist on the stack prior to the beginning of the command-line.

```
." Hello! " CR 3 times
( This prints Hello! 3 times.)
```

Related Words: >IN DO LOOP

TOGGLE (addr mask --)

Does an 8-bit EXCLUSIVE OR of byte at addr with the given mask. Result is left in location addr.

Standards: FIG

```
" fred" COUNT OVER $ 20 TOGGLE TYPE
( change case of 'f' , print  Fred )
```

Related Words: XOR OR

TRACKING (-- var-addr)

See the chapter on CLONE.

TRANSLATOR?

```
TRANSLATOR_LIB
TRANSLATOR_NAME
```

Internal operators to manage the TRANSLATOR library. See :LIBRARY.

Standard: JForth internal .

TRUE (-- true-flag)

A CONSTANT, equal to a boolean TRUE, or -1.

Standards: '83 (79 & FIG use a value of 1)

Related Words: FALSE

TUCK (a b -- b a b)

Duplicate and insert the top item on the stack below the second item. This is a fast coded primitive, the high-level logical equivalent is:

```
11 22 SWAP OVER .S

: NEW-DUMP  ( addr cnt -- )
    TUCK ( save count )
    DUMP  .  ." bytes dumped" ;
```

Related Words: OVER SWAP DDUP

TYPE (addr count --)

TYPE outputs a character string to the EMIT device. The string starts at address ADDR and is COUNT characters long . TYPE will type a maximum number of characters determined by the variable MAX-TYPE to prevent runaway output.

(If you accidentally TYPE something that messes up the screen font, enter a control 'O' to fix it.)

```
" Hello" COUNT TYPE  ( print Hello )
```

Related Words: ID. EMIT COUNT $TYPE ."

TYPEFILE (<filename> --)

Output a file to the current output device. It is similar to the AmigaDOS CLI 'TYPE' command in that it can be paused and restarted from the keyboard. See ?PAUSE.

```
     TYPEFILE S:STARTUP-SEQUENCE  ( see file )
```

Related Words: ?PAUSE FOPEN DOLINES

U* (u1 u2 -- du) " u times"

Same as UM*

Included for compatibility with FIG and '79

U. (n --) "u dot"

Print N as an UNSIGNED number. The number printed will be positive even if the highest bit, the sign bit, is on.

Definition: : U. (n --) 0 D. ;

```
-3 U.  ( prints 4,294,967,293 )
```

Related Words: D.R .HX .R D. .HEX

U/ (du u -- u-rem u-quotient)

```
du = double cell unsigned dividend
 u = unsigned divisor
u-rem = unsigned remainder
u-quotient = unsigned quotient
```

Divide a double unsigned number by an single precision unsigned number leaving an unsigned remainder and quotient.

Standards: Called U/MOD in '79, called UM/ in '83

Related Words: */MOD M/ / M/MOD CELL/ D2/ 2/ W/ /MOD DU2/ U2/

U2* (u -- u*2) "u 2 star"

U2* multiplies the value on the top of the stack by 2 (which is really a left shift). The MSB is shifted out and thrown away. The LSB gets a 0

Standards: JForth unique

Usage: For left shift of bits, or address manipulations and fast math.

Related Words: U* DU2* 2* ASHIFT SHIFT

U2/ (u -- u/2) "u 2 slash"

U2/ shifts the top value on the stack one place to the right. Least significant bit LSB is shifted out and discarded. 0 is shifted in to the MSB position. This is the same as dividing by 2 .

Standards: JForth unique

Usage: When shift right is needed for low level code writing or for dividing by 2.

Related Words: CELL/ D2/ 2/ DU2/ ASHIFT SHIFT

U< (u1 u2 -- flag) " u less than"

Unsigned less-than test. True if U1 < U2, otherwise false.

```
5 -3 .S U<  ( strange but TRUE )
```

Related Words: U. U* U> > <

U> (u1 u2 -- flag) "u greater than"

Unsigned greater-than test. True if U1 > U2, otherwise false .

Related Words: U<

UM* (u1 u2 -- du) "u m times"

UM* inputs two single cell unsigned numbers and returns an unsigned double cell product. From file JU:MULTISTANDARD

Standards: '83, called U* in FIG and '79

M* is to be used if a signed product is needed.

Related Words: M* * */ */MOD U2* 2* DU2*

UM/ (du u -- u-rem u-quotient) "u m slash"

Same as U/

UNMARKFCLOSE (file-pointer --) "un mark f close"

UNMARKFCLOSE is used to remove the file-pointer from the list of files marked to be automatically closed by (QUIT). File-pointers may be added to the list with MARKFCLOSE.

Forth programs usually execute (QUIT) as the final operation in restarting an application, when an error requires restarting (usually indirectly, via QUIT or ?ABORT").

When an application terminates normally, it should use UNMARKFCLOSE to remove the file-pointer from the QUIT list, just prior to closing the file if it was placed there by MARKFCLOSE.

For additional information, see the appendix "JForth File Manager".

Standard: JForth Unique

```
: EXAMPLE ( <filename> -- )
   FOPEN ?DUP          \ get the filename, did it open?
   IF   DUP MYFILE !   \ yes, save pointer in my variable
       MARKFCLOSE      \ set it to auto-close at quit
       RUN-MY-PROGRAM \ do whatever I want with the file
       MYFILE @ DUP    \ need to do TWO thing to the pointer:
       UNMARKFCLOSE    \ remove it from the QUIT list...
       FCLOSE          \ and close the file
   ELSE ." can't open the file "
   THEN
;
```

Related Words: MARKFCLOSE QUIT ABORT ABORT"

UNMARKFREEBLOCK (mem-block --) "un mark free block"

UNMARKFREEBLOCK is used to remove the memory-block from the list of memory-areas marked to be automatically freed by (QUIT). Memory-blocks may be added to the list with MARKFREEBLOCK. When an application terminates normally, it should use UNMARKFREEBLOCK to remove it from the QUIT list just prior to freeing the memory if it was placed there by MARKFREEBLOCK.

For additional information, see the appendix "JForth Memory Manager".

Standard: JForth Unique

```
: EXAMPLE ( - )
   0 1024 ALLOCBLOCK ?DUP \ alloc 1K, did it allocate?
   IF  DUP MYMEM !       \ yes, save pointer in my variable
       MARKFREEBLOCK     \ set it to auto-free at quit
       RUN-MY-PROGRAM    \ do whatever I want to do with the
mem
       MYMEM @ DUP       \ need to do TWO thing to the
pointer...
       UNMARKFREEBLOCK   \ remove it from the QUIT list...
       FREEBLOCK         \ and free the memory
   ELSE ." can't allocate memory "
   THEN
;
```

Related Words: MARKFREEBLOCK QUIT ABORT ABORT"

UNRAVEL (--)

Analyses the return stack and prints a list of the words currently executing. Often used when reporting errors to find out what word caused the error.

```
INCLUDE? UNRAVEL JU:UNRAVEL
: STOPIT  ( -- ) ." Error!" UNRAVEL QUIT ;
: FOO  23 45 + . STOPIT ;
: ZAPPER  ." Test" CR FOO ;
ZAPPER
```

If you do this you should see the names of ZAPPER, FOO and STOPIT in the list. To install UNRAVEL as part of QUIT, use START.UNRAVEL.QUIT. To remove it, use STOP.UNRAVEL.QUIT.

Related Words: R@ RDEPTH >NAME

UNSMUDGE (--)

Clear the smudge bit in the LATEST definition so FIND will locate it. Same as REVEAL.

Standards: JForth

Related Words: SMUDGE REVEAL

UNTIL (flag --)

Will loop back to BEGIN if FLAG = FALSE. Otherwise continue.

Standards: Run time action is '79, '83, FIG.

Compile time action is JForth unique.

```
: YAKYAK
     BEGIN ." Hit key to stop!" CR
         ?TERMINAL   ( leave flag )
```

```
            UNTIL KEY DROP   ( clean up character hit )
        ;
```

At compile time: (address-of-loop-body begin_flag --)

BEGIN_FLAG is for compiler security. ADDRESS-OF-LOOP-BODY is address to branch to if FALSE. UNTIL puts a ?BRANCH into the dictionary.

Related Words: BEGIN WHILE REPEAT UNTIL-NOT END WHILE-NOT DO LOOP AGAIN

UNTIL-NOT until not

Same as UNTIL only stays in the loop under the opposite condition as UNTIL . Simple concatenation of NOT and UNTIL .

Standards: NOT COVERED. JForth EXTENSION.

```
: EXAMPLE  ( -- )
    5 BEGIN  ." loop-body" 1- DUP 0>
      UNTIL-NOT DROP
;
```

Related Words: UNTIL BEGIN

UP! (addr --) "u p store"

UP! is not called in JForth V2.0.

Standards: JForth unique

Related Words: UP@

UP@ (-- addr) "u p fetch"

Fetch the user area pointer address. All user-variables (storage area accessible by only the owning task) are indexed from this JForth-relative base pointer.

Standards: JForth unique

Related Words: USER USP US> >US UP!

UPPER (address count --)

Convert a string at address that is count characters long to upper-case.

```
: GETPASSWORD  ( -- flag )
    BL WORD DUP COUNT UPPER  ( convert mixed case to upper)
    " SECRET-PASSWORD"  $=
;
```

Related Words: MAKEUCASE TEXT=? $= OUTPUT-CASE NOCASE HICASE LOCASE TOUPPER

UPPERC@ (addr -- upper-case-character)

UPPERC@ is used to fetch a character from addr, converting it in the process to upper-case if it is alphabetic. If the ASCII character is not between lower-case a to z, it is not changed. The contents of addr are not affected.

Related Words: C@ UPPER MAKEUCASE LWORD

US-DEPTH (-- user-stack-depth-in-cells) "u s depth"

US-DEPTH returns the number of cells that have been pushed on the user stack, analogous to DEPTH.

Related Words: US> US@ USP US-PICK DEPTH

US-PICK (n -- nth-item-of-user-stack) "u s pick"

US-PICK picks the nth cell off user stack, much like PICK does for the data stack.

Standards: JForth unique.

Related Words: US> US@ USP US-PICK

US> (n --US--) (-- n) "u s from"

This is the user stack equivalent of R> . It pops one item off the user stack onto the data stack .

Standards: JForth unique.

Related Words: USP US> >US INIT-USP

US@ (-- n) (n --US-- n)

US@ copies the top of the user stack onto the stack . This is the user stack equivalent of R@ .

Definition: : US@ (-- N) USP @ @ ;

Standards: JForth unique.

Usage: Control of an extra stack .

Related Words: USP US> >US INIT-USP

USER

Compile time: (name --IN-) Eats name for header from input stream.

Run time: (-- addr) Leaves address of user-owned data area.

USER is a defining word similar to VARIABLE. It defines a data type that is only accessible to the task that created it. This is done by providing each task with it's own USERAREA, from which storage space is allocated.

Each user area contains a counter (called USER#) which is used to point to the next available cell to be allotted. Each time USER is invoked:

1. A name is created in the dictionary, from that which follows the USER declaration.
2. The current value of USER# is fetched, installed into the just-created definition, then incremented by 4 (1 cell) and placed back in USER#.

Note that when the user-variable later executes, it will place the address of the user-specific data area on that users stack. This address will be equal to the user offset stored in the definition, PLUS the address of that users user area (gotten by UP@).

Standards: JForth UNIQUE. FIG USER expects an offset (n -- <name>).

```
USER MY-VAR
23 MY-VAR !
```

Related Words: USER# UP@ #U

USER# (-- user-addr)

This user-variable is used by the USER facility to track the allotment of the user area. See USER.

USERCLEANUP (--)

See the chapter on CLONE.

USP@ (-- addr-of-top-of-user-stack) "u s p fetch"

Returns the address of the top of the user stack, as SP@ does for the data stack.

Standards: JForth unique.

Related Words: USP US> >US INIT-USP

VALID-NAME? (possible-NFA -- seems-valid-flag)

VALID-NAME? takes a possible NFA and returns a true if it appears to be a valid header. It conducts a thorough test on the passed in address, testing if it physically meets the qualities of a real name field.

Standard: JForth Unique.

Related Words: ID. >NAME

VALUE (n <name> --)

Creates a self fetching variable. VALUEs act like a constant except you can change them using the word -> .

```
INCLUDE? VALUE JU:VALUE
765 VALUE MY-VALUE
MY-VALUE .    ( will print 765 )
19 -> MY-VALUE    ( change value )
MY-VALUE .    ( will print 19 )
```

Related Words: -> VARIABLE CONSTANT CREATE { DOES>

VARIABLE (<name> --)

Define a named 32 bit storage location. You can store values into this area for later use. The initial value is zero. When the word is executed it leaves its address on the stack.

Standards: '79 '83. fig expected an initial value for the variable at compile time.

```
VARIABLE MY-VAR
MY-VAR .   ( print address of MY-VAR )
73 MY-VAR  !  ( set value )
MY-VAR @ .    ( prints 73 )
```

Related Words: USER CONSTANT CREATE VALUE

VLINK>VLATEST (voc-link-addr -- addr-of-name)

VLINK>VLATEST converts a VOC-LINK, vocabulary link pointer, address, to the address of the NAME of the latest definition in that vocabulary. See section on Vocabularies for more information.

Standards: JForth UNIQUE

Related Words: VOC-LINK VLINK>VLATEST VLATEST>VLINK

VLINK>' (voc-link-addr -- cfa-of-vocabulary)

VLINK>' converts a VOC-LINK, vocabulary link pointer, address to the CFA address for that vocabulary. See section on Vocabularies for more information.

Standards: JForth UNIQUE

```
VOC-LINK @ VLINK>' >NAME ID.
```

Related Words: VOC-LINK VLINK>VLATEST VLATEST>VLINK

VLATEST>VLINK (nfa-of-vocabulary -- voc-link-addr)

VLATEST>VLINK converts the address of the NAME of a vocabulary to the VOC-LINK address, then vocabulary link pointer address, of that vocabulary. See section on Vocabularies for more information.

Standards: JForth UNIQUE

Related Words: VOC-LINK VLINK>VLATEST VLATEST>VLINK

VLIST (--)

See WORDS.

Related Words: WORDS ALL-WORDS WORDS-LIKE ORDER

VOC-LINK (-- addr)

VOC-LINK is a USER variable pointing to the linked list of vocabularies in reverse order of definition. See section on Vocabularies for more information.

Standards: Not covered . But common in FIG '79 '83 systems.

Related Words: VOCABULARY ORDER WORDS ALL-WORDS VLINK>VLATEST VLINK> VLATEST>VLINK

VOCABULARY (<name> --)

VOCABULARY is a defining word that creates new vocabularies. Vocabularies are used to collect a group of word that relate to a specific area, eg. music, robotics. You can turn on or off a vocabulary to make these words available or unavailable. See section on Vocabularies for more information.

Standard: '83.

Related Words: VOCABULARY ORDER WORDS ALL-WORDS VLINK>VLATEST VLINK> VLATEST>VLINK

W! (n even-addr --) "w store"

Store lower 16 bits of N to an even address.

Related Words: W@ W, W->S

W, (n --) "w comma"

Add the lower 16 bits of N to the dictionary at HERE. Same as , only a 16 bit value.

```
\ Create a table of 16 bit values
CREATE TABLE16  23 W, 497 W, 71 W,
TABLE16 1 2* + W@ .  ( print 497 )
```

Related Words: , ALLOT HERE DP CFA,

W->S (w -- n)

Sign extend a 16 bit value W to a 32 bit value N. See B->S .

```
VARIABLE MY-VAR
HEX -27 MY-VAR W!
MY-VAR W@ .   ( not -27 !!! )
MY-VAR W@ W->S . ( print -27 )
```

Related Words: S->D B->S W@ W!

W/ (n w -- n/w) "w slash"

```
n = dividend ( one cell or 32 bits in size)
w = divisor (one half cell or 16 bits in size)
n/w = 16 bit quotient in lower half of cell
```

32 bit by 16 bit divide. Several times faster than / because it can use the 68000 hardware divide. Does not sign extend 16 bit result n/w . Dividing by zero will cause a trap. The remainder is discarded.

```
DECIMAL  50 4 W/ . ( displays 12 )
```

Related Words: / U/ D/ 2/ ASHIFT /MOD W->S

W@ (even-addr -- w) "w fetch"

Fetch a 16 bit value from the address given. W@ will not sign extend the word. Use W->S to sign extend.

Standards: same as standard @ in most 16 bit Forths.

```
CREATE SHORTVAR -34 W,
SHORTVAR W@  W->S .
```

Related Words: W! W, W->S

WARNING" (flag <string"> --)

Types the quote delimited string if flag is true.

```
: TEST.OVERFLOW
    DEPTH 100 >
    WARNING"  Lots of stuff on stack!" ;
```

Related Words: ABORT" QUIT ."

WHAT'S (<name> -- cfa)

Return the CFA that a deferred word is set to execute. This is useful when you want to set a deferred word temporarily then restore its original value.

Standards: JForth UNIQUE

```
WHAT'S EMIT  >NAME ID.
```

Related Words: IS DEFER GLOBAL-DEFER

WHEN-SCANNED (nfa --)

WHEN-SCANNED is a DEFERed word that is executed by SCAN-VOC and SCAN-ALL-VOCS for each word that they examine. It is used to write WORDS and ALL-WORDS and WORD-LIKE . See SCAN-ALL-VOCS .

Standard: JForth UNIQUE.

Related Words: WHEN-VOC-SCANNED SCAN-VOC SCAN-ALL-VOCS

WHEN-VOC-SCANNED (voc-addr --)

WHEN-VOC-SCANNED is a DEFERed word that is executed by SCAN-ALL-VOCS and SCAN-ALL-VOCS for each vocabulary that is scanned. It is used to write WORDS and ALL-WORDS and WORD-LIKE . See SCAN-ALL-VOCS .

Standard: JForth UNIQUE.

Related Words: WHEN-SCANNED SCAN-VOC SCAN-ALL-VOCS WORDS ALL-WORDS

WHILE (flag --)

Used in the form:

```
BEGIN xxxx WHILE yyyy REPEAT
```

The code XXXX must leave a flag. If the flag is TRUE then YYYY will be executed. Otherwise, execution will jump to the code after REPEAT.

```
: BIGMISTAKE  ( -- , loop while answering yes )
    BEGIN ." Launch a missile?" Y/N
    WHILE ." Bombs away!" cr
    REPEAT ;
```

Related Words: BEGIN REPEAT WHILE-NOT UNTIL

WHILE-NOT (flag --)

Simple concatenation of NOT and WHILE . Instead of

```
BEGIN xxxx NOT WHILE yyyy REPEAT
```

you can use

```
BEGIN xxx WHILE-NOT yyy REPEAT
```

which is faster.

Related Words: BEGIN REPEAT WHILE UNTIL NOT

WILLGET (-- , <modulename> <wordname>)

See the chapter on MODULES.

WINDOWSTRING (-- addr)

Returns the address of the string used to open the JForth window. Change this string and SAVE-FORTH if you want to resize or change the title of the default JForth window.

```
WINDOWSTRING 60 DUMP
```

WITHIN? (N lower upper -- flag)

WITHIN? compares a number to an upper and lower bounds. Returns a TRUE if N is both greater than or equal to lower and less than or equal to upper.

```
20  -5 100 WITHIN? . ( will be true )
20  20 30  WITHIN? . ( true )
20  21 30  WITHIN? . ( false )
```

Related Words: >= <=

WMOVE (from-addr to-addr #words --)

WMOVE is similar to MOVE but it moves n words instead of bytes, and requires the to and from addresses to be even (on the 68000). This is faster than CMOVE.

Standard: JForth unique.

Related Words: MOVE CMOVE CMOVE> $MOVE

WORD (delim-char <text> -- $addr)

Reads one word from the input stream using the character on the stack as the delimiter. An ASCII blank is usually used. WORD moves the text string <TEXT> to the dictionary address $ADDR with the count in the first byte, and leaves $ADDR on top of the stack. $ADDR typically equals HERE. WORD will ignore leading occurrences of the delimiter character.

WORD will update the TIB pointer >IN to point to the character following the text.

If SKIP-WORD has been called then WORD will ignore the following text and simply return the address of HERE . This can be used to pass a string to a word that normally gets its text from input. See SKIP-WORD for more info.

Here is an example of using WORD to define the comment operator.

```
: (  ASCII ) WORD DROP ;
```

Related Words: HERE EXPECT TEXT KEY UNWORD >IN FIND SKIP-WORD

WORD-SWAP (abcd -- cdab)

Swap upper and lower 16 bit parts of a 32 bit number. A B C and D in the diagram are bytes.

```
HEX 12345678 WORD-SWAP .  ( print 56781234 )
```

Standards: JForth unique

Related Words: W@ W! BYTE-SWAP

WORDS (--)

WORDS causes a printout to the current device of all JForth words in the CONTEXT vocabulary.

Standards: '83. Called VLIST in FIG and '79 . Both are included in JForth.

Related Words: ALL-WORDS VLIST ORDER VOCABULARY SCAN-VOCS

WORDS-LIKE (<partial-name> --)

Searches dictionary for words whose name contains the partial name given.

```
WORDS-LIKE EMIT
```

will print FEMIT FLUSHEMIT (EMIT) plus any other words containing "EMIT". WL is a short alias for WORDS-LIKE. WORDS-LIKE is also attached to the function key <F6>.

Related Words: FILE? WORDS

X! (nx ... n2 n1 addr x --)

Store x cells from stack to memory starting at address ADDR. Cells are transferred so that the most significant cell, nx, is at addr, with less significant cells in ascending memory. Works on ODD or EVEN addresses.

Standards: JForth UNIQUE

```
VARIABLE MY-DVAR CELL ALLOT
9. MY-DVAR 2 X!  (  stores the double number 9. in MY-DVAR )
```

Related Words: X@ X>R XR> D!

X>R (nx...n2 n1 x --) (--R-- nx...n2 n1)

Transfers x cells from the data stack to the return stack .

Standards: JForth UNIQUE.

```
: EXAMPLE  2  46 88 99  3 X>R  . ( displays 2 )
    3 XR>  3   XDROP ( empty stack ) ;
```

Related Words: XR> XR@ XDROP XDUP

X@ (addr x -- nx...n2 n1) "x fetch"

Fetch x cells (nx...n2 n1) from memory starting at address ADDR to the data stack. ADDR is address of nx and addresses are incremented by one cell address space to fetch each new N. See X! .

Standards: JForth UNIQUE

```
CREATE FOO  11 , 22 ,  33 ,
FOO 3 X@  .S  ( 33 22 11 on stack )
```

Related Words: XR> X>R XR! XDUP XDROP X!

XDROP (nx...n2 n1 x --)

Drop X cells from the data stack. This is very fast regardless of the number of items dropped.

Standards: JForth UNIQUE

```
11 22 33 44   4 XDROP  ( empty stack )
```

Related Words: XR> X.R XR@ XDUP

XDUP (nx ...n2 n1 x -- nx...n2 n1 nx...n2 n1)

Duplicate the top x cells on the stack in order and push the whole set on top of the stack .

Standards: JForth UNIQUE

```
11 22 33  3 XDUP .S ( print: 11 22 33 11 22 33 )
```

Related Words: XR> X>R XR@ XDROP

XOR (n1 n2 -- n3)

Perform bit by bit EXCLUSIVE OR of N1 and N2. Here is the truth table for an EXCLUSIVE OR operation on two bits:

```
A   B  ->  A.XOR.B
0   0          0
1   0          1
0   1          1
1   1          0
```

(Performing an XOR with -1 has the effect of reversing the value of the bits in a number.)

Standards: '79 '83 fig

```
BINARY 0011 1010 XOR .  ( print 1001 )
010101 111111 XOR . ( print 101010 )
```

Related Words: OR AND NOT

XPICK (..... N X -- Xcells-from-Ncell-deep)

XPICK takes the cells N deep and the number of cells X to pick and picks them from the stack. Zero is the 1st cell.

Standards: JForth UNIQUE

```
99 88 77 66  (  -- 99 88 77 66 )
1 2 XPICK .S (  -- 99 88 77 66 88 77 )
```

Related Words: R> >R X>R XR@ XDROP XDUP PICK RPICK

XR> (x -- nx ...n2 n1 n0)

```
( nx ...n2 n1 n0 --R-- )   "x r move"
```

Move x cells (n1 n2 up through nx) from the return stack to the data stack in order.

Standards: JForth UNIQUE

Related Words: R> >R X>R XR@ XDROP XDUP

XRDROP (x --) "x r drop"

XRDROP drops X items off the return stack.

Standards: JForth UNIQUE

```
: EXAMPLE   4 >R  38 >R  30 >R   3 XRDROP ;
```

Related Words: R> >R X>R XR@ XDROP XDUP

Y/N (-- flag , or quit) "y slash n"

Y/N is a word that displays the message "...Yes, No or Quit (y/n/q)?" then waits for 1 of 3 keys: y n or q in either upper or lower case. If q it will call QUIT, if y it will return a true flag, and if n it will return a false flag (zero).

Standards: JForth UNIQUE

[(--) "open bracket"

Suspends the operation of the compiler, causing following input to be INTERPRETed, rather than compiled. Usually paired with the] character, which re-invokes the compiler.

Standard: fig '79 '83

Here is an example where the calculation of a constant is done at compile time to reduce the run time burden.

```
: EXAMPLE   [ HEX 7A43 17 / DECIMAL ] LITERAL +
   [ ." Prints right now!!" ] ;
```

Related Words:] STATE IMMEDIATE INTERPRETING? COMPILING? LITERAL

[COMPILE] (<name> --) "bracket compile"

Forces the compilation of the next word in the input stream even if it is an IMMEDIATE word.

```
: VERBOSE.LITERAL   ( N -- , compile N )
    DUP ." N = " . CR
    [COMPILE] LITERAL  ( execute LITERAL later )
; IMMEDIATE
: FOO [ 234 9 / ] VERBOSE.LITERAL . ;
FOO .
```

Related Words: [] CFA

[] (object --)

Used to indicate late binding in ODE. See chapter on ODE.

```
: PRINT.OBJECT ( object -- )
    PRINT: [] ;
```

Note: This is also sometimes used as a shorthand for [COMPILE] whose use is preferred.

Related Words: [COMPILE]

\ (<rest-of-line> --) "back slash"

Ignores the rest of the line during compiling or interpreting. Treat it as a comment.

```
\ This is a comment.
." DO THIS!"  \ But not this!
```

Related Words: ( )

] (--) "close bracket"

Enter compile mode by setting STATE to TRUE. Used with [to temporarily suspend and resume compilation.

This is sometimes entered accidentally because of it's proximity to the carriage return key. Just enter [to get back your OK prompt.

Standard: fig '79 '83

```
: example   [ 2 4 + . ]  [ 9 ] literal . ;
\ Will display 6 when compiled and 9 when run.
```

Related Words: [STATE IMMEDIATE INTERPRETING? COMPILING?

Appendices

Appendix A
Delta Research Biographies

Mike Haas

Mike first fell in love with Forth when he ported FIG Forth to a homebrew 6800 system. He worked with Brian and Jim on the design of a Forth chip and is also a 'C' programmer. Mike is also working on networking software for the Macintosh. His work on JForth includes the assembly language kernal (the foundation of JForth), the Amiga system interface, CLONE, Modules, the CALL facility that makes the Amiga libraries accessible, and more. Mike also developed Textra, the text editor for programmers, and LCDCalc, an attractive calculator that looks like a desktop LCD model.

Phil Burk

Phil became a Forth fan developing code for a music system running on a, then state of the art, S-100 68000 system. He has developed a Forth for the VAX and for a homebrew 68000 computer. He worked first as a physicist, then later developed interactive graphics/mapping systems on mainframes using 'C' and FORTRAN. His hardware projects have included small custom computers, guitar to MIDI convertors, and synthesizer modules. His work on JForth includes structures, ODE , the graphics toolboxes, the '.j' files, Hashing, History, Source Level Debugger, IFF, the new local variables, and many demos. Phil also works on **HMSL**, the Hierarchical Music Specification Language. HMSL is an experimental composition language based on JForth. One of Phil's latest interests is in developing real time sound synthesis software using the 56000 Digital Signal Processor.

Brian Donovan

Brian first became interested in Forth when he saw a complete 8080 macro assembler contained within a 1/4 page footnote in an IEEE magazine. He has been programming for about 10 years doing mostly Forth, assembly and 'C'. His hardware projects have included electromechanical games, electric vehicles, telephone network testers, smart refrigerators and the design of a custom Forth processor. Brian has also worked with Novix on their Forth processor. His work on JForth includes part of the kernal, the original local variables, vocabularies, WORDS-LIKE, multistandard, and more.

Jim King

Jim is a chemist, a musician, an attorney, and an electrical engineer. He is also a 'C' and Forth programmer. He has worked with Mike and Brian on the Forth chip, and other hardware projects including audio digital signal processors. He helped put together the manual, and did quality assurance.

Appendix B
Resources

On-Line

These resources are accessable via modem. Due to the volatile nature of cyberspace, these resources may not be available in the future.

GENIE

GEnie is a national bulletin board run by General Electric. They maintain a very active Forth Roundtable. Enter M710 or FORTH. Information available by calling (800)638-9636. GEnie charges an hourly rate.

BIX

Martin Kees hosts a JForth vendor conference on BIX. For information, call (800)227-2983.

Commotech Computers BBS

Commotech Computer store in San Diego has a JForth section on their BBS. To get to the JForth stuff, go to the Mail Menu, then select "Other Section". They also have some JForth related files. If you use JRComm, set it for 16 color ANSI/IBM mode. The phone number is: (619)477-2368.

HMSL BBS

This is a free BBS intended for HMSL users, but it has a thread for the Amiga that has lots of JForth discussion. Hosted by Robert Marsanyi. Lots of high level electronic and computer music stuff. Call (415)928-8246 but don't hog it.

InterNet

You can reach the authors, Phil Burk and Mike Haas, via the internet. Send electronic mail to:

```
phil@mills.edu
   (or)
haas@starnine.com
```

Organizations

There are two major sources for information on FORTH. They are:

```
FORTH Interest Group
P.O. Box 8231
San Jose, CA 95155
(408) 277-0668
Fax: (408) 286-8988
```

FIG is a worldwide FORTH users group. They publish FORTH Dimensions, an excellent journal of new ideas in FORTH. Membership is highly recommended. *Get a copy of their order form* which has almost every Forth related book and periodical including the Dr. Dobbs FORTH issues, and the publications of the Institute below.

```
The Institute for Applied FORTH Research, Inc.
70 Elmwood Avenue
Rochester, New York, 14611
```

The Institute publishes the Journal of FORTH Applications and Research. This is a technically oriented publication. They also organize an annual conference in Rochester.

Publications

1983. *FORTH-83 Standard.* A Publication of the FORTH Standards Team. Mountain View Press. Mountain View, CA.

Brodie, Leo. 1981. *Starting FORTH.* Prentice Hall, Inc. Englewood Cliffs, New Jersey 07632. A good introduction to FORTH. It's emphasis is on polyFORTH, an older 16 bit FORTH, so not everything applies to JForth. Easy reading for beginners.

Burk, Polansky, Rosenboom. 1986. *HMSL - Hierarchical Music Specification Language - Reference and User Manual,* Frog Peak Music, Box 151051, San Rafael, CA, 94915-1051. HMSL is an experimental music composition language written in JForth and ODE.

Haydon, Glen B. 1983. *All about FORTH, An Annotated Glossary.* Mountain View Press. Mountain View, CA. This provides source, and a description of the words in MVP FORTH.

Kelly, Mahlon G. and Spies, Nicholas. 1986. *Forth: A Text and Reference.* Prentice-Hall Software Series, Englewood Cliffs, New Jersey 07632. A comprehensive tutorial and reference manual that addresses some of the differences between FORTHs. The emphasis is on MMSFORTH for the IBM PC.

Chirlian, P. 1983. *Beginning FORTH.* Matrix. Beaverton, Oregon. A good introduction for beginners.

Derick, M. and Baker, Linda. 1982. *FORTH Encyclopedia.* Mountain View Press. Mountain View, CA. Provides source code and a detailed breakdown of the words in an 8080 version of FIG-FORTH. Handy for people who want to dig into the internals of a FORTH.

Hofert, D. 1985. *A Bibliography of FORTH References,* 2nd ed. The Institute for Applied FORTH Research, Inc. Rochester, New York. A list of most articles published on FORTH, cross referenced.

Kane, Hawkins and Leventhal. 1981. *68000 Assembly Language Programming.* OSBORNE/McGraw Hill, Berkeley, CA. Excellent book for learning 68000 assembly with many examples. An extensive reference section that covers every instruction in depth also makes this an excellent reference. Recommended.

Loeliger, R. 1981. *Threaded Interpretive Languages.* Byte Books, Peterborough, N.H. A hackers book that describes the implementation of a TIL on a Z80. The author implies that is not FORTH but it sure looks like it to me. Good description of traditional inner interpreters. JForth doesn't have an inner interpreter so this is not directly relevant.

Winfield, A. 1983 *The Complete FORTH.* Sigma/Wiley, New York. Excellent text on FORTH-79. Little advanced material.

Object-Oriented Programming

BYTE Magazine, Volume 11, Number 8, 1986, "Object Oriented Languages" issue; contains articles on object oriented Forth, NEON, ACTOR, Objective-C, Smalltalk, etc.

Cox, Brad J., *Object Oriented Programming: An Evolutionary Approach,* Addison Wesley Publishing Company, 1986

Goldberg, Adele, and Robinson, David, *Smalltalk-80: The Language and Its Implementation*, Addison Wesley Publishing Company, 1983

ASCII Control Characters

Decimal	Hex	Character	Key	Typical Usage
0	00	NUL	^@	Denotes end of string or no data.
1	01	SOH	^A	
2	02	STX	^B	
3	03	ETX	^C	
4	04	EOT	^D	
5	05	ENQ	^E	
6	06	ACK	^F	ACKNOWLEDGEs reciept of message.
7	07	BEL	^G	Rings BELL on printer or terminal.
8	08	BS	^H	BACKSPACEs cursor.
9	09	HT	^I	Horizontal TAB.
10	0A	LF	^J	LINE FEED, AMIGA line terminator.
11	0B	VT	^K	Vertical TAB.
12	0C	FF	^L	FORM FEED, advance to next page.
13	0D	CR	^M	CARRIAGE RETURN, terminate line input.
14	0E	SO	^N	
15	0F	SI	^O	
16	10	DLE	^P	
17	11	DC1	^Q	Resume host transmission, XON.
18	12	DC2	^R	
19	13	DC3	^S	Stop host transmission, XOFF.
20	14	DC4	^T	
21	15	NAK	^U	NOT AcKnowledged, NOT recieved.
22	16	SYN	^V	
23	17	ETB	^W	
24	18	CAN	^X	
25	19	EM	^Y	
26	1A	SUB	^Z	
27	1B	ESC	^[	ESCAPE, general purpose, change mode.
28	1C	FS	^\	
29	1D	GS	^]	
30	1E	RS	^^	
31	1F	US	^_	

Printable Characters -- (Decimal Hex Character)

32	20	SPAC	48	30	0	64	40	@	80	50	P	96	60	`	112 70 p
33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113 71 q
34	22	"	50	32	2	66	42	B	82	52	R	98	62	b	114 72 r
35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115 73 s
36	24	$	52	34	4	68	44	D	84	54	T	100	64	d	116 74 t
37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117 75 u
38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118 76 v
39	27	'	55	37	7	71	47	G	87	57	W	103	67	g	119 77 w
40	28	(	56	38	8	72	48	H	88	58	X	104	68	h	120 78 x
41	29	)	57	39	9	73	49	I	89	59	Y	105	69	i	121 79 y
42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122 7A z

43	2B	+	59	3B	;	75	4B	K	91	5B	[
44	2C	,	60	3C	<	76	4C	L	92	5C	\
45	2D	-	61	3D	=	77	4D	M	93	5D	]
46	2E	.	62	3E	>	78	4E	N	94	5E	^
47	2F	/	63	3F	?	79	4F	O	95	5F	_

107	6B	k	123	7B	{
108	6C	l	124	7C	|
109	6D	m	125	7D	}
110	6E	n	126	7E	~
111	6F	o	127	7F	DEL

Appendix D
Sample Applications

We have provided several complete applications written in JForth. They can be used either directly from JForth or they can be cloned and used as CLI commands. Some of them demonstrate how to parse the command line, how to handle input errors and print "usage" information. The file names all end with ".f" to distinguish the Forth source file from the cloned image file. These files can be found on the "JA:" logical volume.

Please see the chapter on Clone for instructions on how to clone these applications.

CR2LF.f - Carriage Return to Line Feed

Converts every Carriage Return in a file to a Line Feed. This is handy if you get a file from some unfortunate Macintosh user.

DIAL.f - Quick Dialer using a Modem

DIAL is a CLI based utility that will look up and dial telephone numbers through your modem. All you have to do is enter in a CLI:

```
DIAL FRED
```

then pick up the phone and wait for Fred to answer. DIAL will search a text file containing names and phone numbers. When it finds a match it opens the SER: device and send a Hayes compatible message to your modem to dial. DIAL was written as a shell command because a mouse driven interface would take longer to use then just dialing the phone directly.

To install DIAL you must:

1) Copy DIAL to your C: directory or other command directory.

2) Set your serial preferences to match your modem. Eg. 1200 baud, 8 data bits, etc.

3) Create a file called S:PHONELIST containing names and phone numbers. Names and phone numbers cannot have spaces inside them. For example:

```
JOE  1(707)555-1212                <-- GOOD
FRED FLINTSTONE 1 (707) 555-1234   <-- BAD
```

There are two ways to do this. One is using a text editor. The other is to use the ADD option of DIAL. For example:

```
DIAL ADD  WILMA  1(415)555-9876
```

The above will add WILMA's name to your phone list. If the phonelist file does not exist, DIAL will create it. To delete names, or to change names, just edit the file using a normal text editor like EMACS or Textra. You can optionally specify a filename other than S:PHONELIST. For example:

```
DIAL IRMA MYSTUFF:MYLIST
```

Since that would be more typing than just dialing it yourself, I recommend making an alias as follows:

```
ALIAS  DL DIAL [] MYSTUFF:MYLIST
```

Put that alias in your S:SHELL-STARTUP file.

Note that DIAL.f is shareware. If you bought JForth then ignore that. Consider yourself paid up. Please feel free to give a cloned version of DIAL to a friend. Just make sure you also give the Dial.README file which contains this documentation and information on the shareware aspects of Dial.

Docu.f - Automatic "Documentation" Generator

This application can be used to automatically generate simple documentation from a file. It scans a file and prints any line that has a character in the first column, is not a comment, and has a stack diagram on it. Thus the following lines would be printed if they occurred in a file.

```
: ADD1  ( N -- N+1 , add one to a number )
and
VARIABLE NUM-FOO ( -- addr , number of foos)
and
: ADDEM  { aa bb -- aa+bb , add em up }
```

The word which should be cloned is:

```
: DOCU.FILE ( <filename> -- , document file )
```

DumpBrush.f - Dump a brush as JForth source code.

This application reads an IFF file containing a brush and outputs JForth compatible source code. Thus you can use DPaint to create images and use them in your programs as gadget images. Clone this program then enter:

```
DumpBrush >outfile brushfile
```

DumpIFF.f - Dump IFF file for analysis.

This scans an IFF file and prints out the contents of the chunks it encounters.

H2J.f - Convert a 'C' style ".h" file to a ".j" file.

This is the program we used to convert the Amiga DOS include files from 'C' code to JForth compatible code. See the chapter on Amiga Libraries and Structures for more information.

Print.f - Print a File

PRINT will output a file to a printer. It prints the file name, page number, and date on each page. It will also put line numbers on each line and skip over page perforations. PRINT uses the left and right margins set in preferences to tell whether a line will wrap around. Thus, lines that wrap will not mess up the page breaks. You can turn off the line numbers using the "-n" option. You can set the number of spaces for a tab using the "-t#" option. The default tab spacing is 8 spaces between tabs. To print a file called MYSOURCE without line numbers and with tabs set to 6 spaces, enter:

```
PRINT MYSOURCE -N -T6
```

If you want to send the output to a file instead of the printer, give that filename after the source file. For example:

```
PRINT MYSOURCE OUTFILE
```

Rude.f - Print Rude Message using an Alert

You can use this application to scare your friends or to control command files. RUDE will read a quoted string from the command line and display it in an Alert just like a GURU Meditation Error. The sight of these messages is enough to send a chill down the spine of most Amiga users.

Rude asks you to hit the left button for Yes and the right for No. A Yes will return 5 which can then be used to control the command file. Here is a sample Amiga DOS command sequence (not Forth).

```
RUDE "Should I install TextMangler 3.4?"
IF WARN
        RUN TextMangler
ENDIF
```

SortMerge.f - Merge Presorted Files

SORTMERGE will take two presorted files and merge the output into a third file in sorted order. This application is useful for adding new data to a large presorted file. It can be handy when used with DOCU.FILE to create quicky documentation of a large multifile project. Use the Amiga DOS SORT command to presort the input files.

```
SORTMERGE infile1 infile2 outfile
```

SayNumber.f - Recite a single-precision number in decimal

This JForth program parses the next word in the input stream and pronounces it in decimal-quantified form. For example, the "Say" command, as furnished by Commodore, would pronounce 123 as "one, two, three", whereas this program will say "one hundred and twenty-three".

```
saynum 123
saynum 1989
```

Terminal.f - Very Dumb Terminal Program

This very simple terminal program can be used as a starting point for a fancier terminal program.

Update.f - Copy newer files from one directory to another

This JForth program examines two directories, then copies everything from <srcdir> to <destdir> that meet the specified criteria. This is useful for maintaining backups of projects on floppies. UPDATE was written by Mike Haas for us at Delta Research as we worked on JForth on multiple Amigas. We could merge our work by updating to a central computer.

Usage:

Update SourceDir DestDir [-date|-size -list -noask -both]

```
-date = SourceDir/file is later than DestDir/file (default)
-size = SourceDir/file is different size than DestDir/file
-list = Don't really copy, just list the files that WOULD.
-noask= Don't ask the user to verify input
-both = Update only those files that exist in BOTH directories
-all  = Check all files in the directory and any subdirectories.
```

Only the first letter of the option is required. Here is an example that updates everything from a hard disk resident volume called MYWORK: to a floppy in DF1:.

```
UPDATE MYWORK: DF1: -N -A
```

WordCount.f - Count Words Lines and Chars

This is very similar to the UNIX wc command except it gives you all the information at once. The main word is called WC and is used like this:

```
WC filename
```

Appendix E
Style Guidelines

Naming Conventions

Forth allows ultimate flexibility in the naming of words. For this reason, it is very important that the programmer be consistent in their naming. Naming conventions can provide a road map to your code. If you ever have to go back and revisit code months, or years, after writing it, you will be thankful if you used systematic naming.

This section will provide some of the naming conventions used in JForth and offer suggestions for your own naming. Since JForth naming is the combination of naming conventions from 3 developers, plus the standard Forth names, you may notice deviation from these suggested standards in JForth. These suggestions will be more rigidly adhered to in the code I wrote since I often take my own advice. Here goes:

Use Prefixes to Mark Related Code

By a 2 or 3 letter prefix to all the words of a given system, you can clearly identify those words when they appear in other code. This also helps eliminate naming conflicts if the prefixes are unique. Examples in JForth are:

```
GRxxx - for GRaphics words.
WD_xxx - for members of the WinDow structure
DBxxx - for internal Debugger Words
$xxx - for string related words.
```

Use Signature Characters to indicate a word's function.

You can often place characters in the name that indicate what a given word does. Examples are:

```
4+ - adds 4
CONSOLE!  - set pointer in CONSOLE variables.
GR.COLOR!  - set graphics color (store in RastPort)
```

(xxx) - parentheses usually indicate an internal function that is called from a deferred word, or that is compiled for run time action. Sometimes <> and [] are used. Examples are:

```
(EMIT)  - simplest EMIT word.
(($"))  - run time action of ".
```

{ and } are handy for marking words that are balanced. An example is:

```
DEBUG{  : FOO DUP + ; }DEBUG
```

Use Separators that indicate the Type of Word.

Compound words can use a separator that give you some idea of what the word is. The separators I use are:

- "hyphen" for variables, objects, and other data structures that leave an **address** on the stack. These will generally be followed by a @ or !. Examples are:

```
HIGHLIGHT-INPUT    GR-CURWINDOW    MAX-INLINE DL-LINENUM
```

_ "underscore" for constants, values, or other words that leave their actual **value** on the stack. Examples are:

```
OFFSET_BEGINNING    KH_HISTORY_SIZE    MEMF_CLEAR
```

. "dot" for action words, verbs. Examples are:

```
GR.DRAW    DEBUG.START    PIC.LOAD
```

Use full names for obscure words, short names for common words.

Short names are good for common words because no one likes to type long names all the time. For rarely used words, however, you should use long names because you are less likely to remember what they do. There is also less likelihood of naming conflicts with long names. Remember when an application is cloned the names are removed and do not add size to the final image.

Writing Style

Here are some random thoughts on programming style.

Beware of compile time initialization.

If your program needs to open files, allocate memory, build jump tables or do anything with addresses, please put that code in a colon definition. And please do not call that word in the file itself. Doing so might seem like a handy little trick but it will bite you if you do a SAVE-FORTH or try to CLONE your program. Opening files, and allocating memory generate addresses that are only valid for that time. If you do a SAVE-FORTH or CLONE you may save an address that will not be valid the next time you use it.

Here is an example of some BAD code in a file:

```
VARIABLE MYFILE
VARIABLE MYMEM
FOPEN RAM:DATA MYFILE !  \ Bad! Bad programmer!
MEMF_CLEAR 2000 ALLOCBLOCK MYMEM !  \ Very Bad!
```

Here is an example of doing it right. Notice that we also check for error flags on initialization, and provide for automatic if we forget the code. Using these techniques will save you from many puzzling errors.

```
VARIABLE MYFILE
VARIABLE MYMEM
: DOITRIGHT  ( -- error? , "Good programmer! Here bisquit.")
   TRUE \ default error return
   " RAM:DATA" $FOPEN ?DUP  \ did it work
   IF
      MYFILE !
      MEMF_CLEAR 2000 ALLOCBLOCK ?DUP
      IF
         MYMEM !
```

```
            DROP FALSE  \ change error?
          THEN
        THEN
    ;
    : CLEANUP ( -- )
        MYFILE FCLOSEVAR
        MYMEM FREEVAR
    ;
    IF.FORGOTTEN CLEANUP
```

Write short words.

Chuck Moore first pointed out that short definitions give you the most flexibility. Nothing is worse than a word that does TOO much. If you want part , but not all, of what a word does you cannot use it. It is useless. But if a word does only part of what you need you can add the rest.

Place Stack Diagrams on Every Word.

and sometimes inside words. It is nearly impossible to read someone else's code if there are no stack diagrams. I have also seen people trying to write a word when they don't know what its stack diagram should be, another impossible task.

Indent Paired Words to the Same Level.

This really helps improve the readability of code and can prevent many common mistake. I have come to believe that IF, ELSE ,THEN, BEGIN, WHILE, REPEAT, UNTIL, CASE and ENDCASE should always be on their own line and cause a change in indentation level. Here is an example of the suggested indentation style.

```
: FOO ( a -- b , do something )
    DUP 21 <  ( -- a flag, generate flag on previous line
    IF
        0  ( -- a 0 )
        DO  I . CR
        LOOP ." All done!" CR
    ELSE  ( a -- , good place for stack diagram )
        100 >
        IF  ." A really big!" CR
        THEN
    THEN
;
```

Use >R R@ R> or Local Variables -

to avoid excessive stack dancing. Too much DUP SWAP DROP will ROT your brain.

Write your Code Backwards using "Top Down" programming.

Start by writing the top word in your application using an English like set of words. Once this makes sense, write the next lower level, and so on. Forth is enough like English that your "pseudo-code" can end up being legal Forth by the time you are done. Forth is typically thought of as a "bottom up" language because it is so easy to hack in. This is fine for pure experimentation but big projects demand the discipline of "top down" coding.

Initialization, Action, and Termination

Divide each module of your program into three parts: Initialization, Action, and Termination. This will greatly simplify debugging because you will be able to completely initialize the system and then examine it before taking any action. It also helps to organize your code. Initialization typically consists of things like opening files, allocating memory, opening windows, or creating gadgets.

Transportability Techniques

by Brian Donovan

This appendix describes some techniques to use for writing transportable high level Forth code.

The CELL concept.

In Forth programs, you often want to skip over a single precision unit of data, or several single precision units of data, called "cell"s in Forth. Until recently, most Forth programmers were explicitly using the numeric count needed on the machine they happened to be using. This meant that on a 16 bit 8086 Forth they would use a 2+, on a 68000 32 bit Forth, like JForth, they would use 4+ , and on a NOVIX Forth CPU with word addressing, they would use 1+ . For subtraction, multiplication and division by the single precision unit addressing size, the same situation would arise, thus, all programs written that way would be totally untransportable. A very simple and low overhead solution to this problem is to use CELL+ CELLS CELL/ CELL- and CELL instead of 1+ 2+ 4* etc.. whenever the program is working with cells. We have painlessly used the same source for 16 bit and 32 bit computers of many types, and have found that programs written with CELL are much less obscure to read.

In JForth: CELL = 4

One way to automatically assign CELL size in your programs is as follows:

```
SP@ SP@ - ABS CONSTANT CELL
: CELL+  CELL + ;
: CELL-  CELL - ;
: CELLS  CELL * ;
: CELL/  CELL / ;
```

With conditional compilation, you can pick out the optimal speed words for functions like CELLS and CELL/ . In JForth, all these words are machine coded.

INLINE concept.

In more advanced Forth programs, programmers often want to create new inline data words (like ."). With the advent of segment threaded Forth, relative address Forths (like JForth) , and other new ways of implementing Forth, The top return stack item does not necessarily point data following the called word (inline data). We use a slow version of LIT to illustrate some of the problems.

This is slow LIT the way it would be in absolute addressing, address threaded Forth (the good old standard unfancy Forth on a Z80 for example.)

```
: SLOW-LIT  ( --- N )  ( CELL --INLINE-- )
    R@ @ R> CELL+ >R ;
```

SLOW-LIT on a relative addressing Forth (like JForth) :

```
: SLOW-LIT   R@ >ABS @ R> CELL+ >R ;
```

SLOW-LIT on some segment threaded 32 BIT, 8086 Forths:

```
: SLOW-LIT-SEG   R@  SEG+OFF>ADDR  @
    R> SEG+OFF>ADDR CELL+ ADDR>SEG+OFF >R ;
```

...pretty bad ? You would have to rewrite any section that uses inline data, when you changed Forths. You can avoid this problem by using the INLINE words: INLINE+ INLINE@ >INLINE and INLINE> (and adding them to other systems, as appropriate.) you can write fully transportable inline code:

```
: SLOW-LIT  ( --- N )  INLINE@ @ CELL INLINE+ ;
```

This definition will work on ALL of the Forths I have over encountered, as long as you add the appropriate 4 inline definitions to the dictionary first. by the way, SLOW-LIT is used as follows:

```
: SLOW-LITERAL   ( N --- )
    COMPILE SLOW-LIT , ; IMMEDIATE
: EXAMPLE  [ 5 ] SLOW-LITERAL ;
```

You can always add these definitions to a system that doesn't have them. The definitions are very implementation dependent. For an old fig Forth system they would be the simplest:

```
: INLINE+  ( N --- )  ( ADDR --R-- ADDR+N )
    COMPILE R> COMPILE + COMPILE >R ; IMMEDIATE
: INLINE>  COMPILE R> ; IMMEDIATE
: >INLINE  COMPILE >R ; IMMEDIATE
: INLINE@  COMPILE R  ; IMMEDIATE
```

In JForth the definitions are slightly more complicated, and on some of the 32 bit segment threaded Forths, they are fairly ugly. But once written, all the rest of your code flies!

You may be confused by a word in JForth, INLINE , which has little to do with the inline words above. INLINE sets a flag in a words header, telling the compiler that this word must be compiled inline, it cannot be called (for instance >R) .

Don't Use JForth Internal Words

Avoid using JForth internal words unless you can reconstruct them from scratch.

Understand Unique Features Before Using Them

Unfortunately, you also have to avoid using some of the nice JForth utilities unless you can reconstruct them as well. At least we will keep the JForth utilities around. All of the JForth unique programs are copyrighted, but many are available for distribution without charge. If the file does not specifically state, that it may be freely distributed, than you must get our permission to distribute it elsewhere. We encourage you to take the CELL and INLINE concepts and programs anywhere you want. Many of the JForth utilities are available on other Forths as well, and we made every attempt at compatibility with other Forths where this is appropriate. Some programs that you need permission to distribute are: ODE ASM DISM MULTISTANDARD LOCALS

Delta Research is actively helping to establish better standards, particularly for high-end Forth systems.

Conditional Compilation.

Use the conditional compilation words, .IF .ELSE .THEN .NEED INCLUDE? , to adapt to different systems automatically. Usually the differences between systems are small enough to allow one source

to deal with all target systems. Major sections that are different between targets, can be put into separate files, and conditionally loaded with INCLUDE? . Unfortunately, The Forth community has not standardized the names for the conditional compilation words. JForth uses the same names as LMI Forth. (The '83 standard include a SUGESTED set of names that we think are unacceptable: IFTRUE OTHERWISE IFEND). The words we have used are easy to remember, since they work like the familiar IF ELSE THEN words all Forth programmers know so well. The "." sets them apart from the ordinary conditionals as well. We would have preferred [IF] [ELSE] [THEN] to emphasize the fact that these words work in a different state, but we felt it was better to use an existing acceptable set of words.

Multistandards

If you need to compile code that conforms to the FIG, Forth'79, or Forth'83 standard, then INCLUDE the file JU:MULTISTANDARDS.

Apendix F
Words by Function

Address Conversion

```
>ABS >REL IF>ABS IF>REL CALL>ABS
```

AMIGA Library Access

```
ARGS CALL CALL>ABS CALLVOID CALLVOID>ABS CLOSEALLLIBS CLOSELIB
DCALL LIB? LIBRARY LIBVERSION OPENLIB
```

Arithmetic

```
* */ */MOD + +! +- - / /MOD 1+ 1- 2* 2+ 2- 2/ 4* 4+ 4- 4/ << @BITS
ABS ASHIFT BIT-SET? BYTE-SWAP CLR-BIT COMP D+ D- D2* D2/ DABS
DNEGATE DU2* DU2/ EVEN-UP FIG-NOT M* M/ M/MOD MAX MIN MOD NEGATE
SQRT U* U/ U2* U2/ W/
```

Conditional Compiling

```
.ELSE .IF .NEED .THEN EXISTS? INCLUDE?
```

Conditional Operators

```
0< 0= 0> < <= = > >= D< D= DU< FALSE NOT TRUE U< U> WITHIN?
```

'C' Structures from JU:CSTRUCT

```
..! ..@ :STRUCT ;STRUCT APTR ARRAYOF BYTE BYTES DST GETMODULE LONG
OB.STATS? S@ S! SHORT SIZEOF() STRUCT UBYTE ULONG
UNION{ USHORT }UNION }UNION{
```

Compiler Support

```
!CSP >INLINE >PARENT ?COMP ?CSP ?EXEC ?PAIRS BOTH BSR-CODE
CALLADR, CARRAY CFA, COMPILE COMPILING? CREATECHAR DLIT DLITERAL
DO-DOES-SIZE DOES> IMMEDIATE INLINE INLINEOK? INTERPRETING? JSR-
CODE LIT LITERAL LONGCFA, RECOGNIZE RTS-CODE SMUDGE STATE UNSMUDGE
[ [COMPILE] ]
```

Debugging Aids

```
&CHARS .FROM .FROM-OR.  .VOC 1010-EMULATE 1111-EMULATE }DEBUG
BEST-GUESS DEBUG DEBUG{ DEBUG.ON DEF DO-TRACE DST DUMP FILE? FIND-
DATA FIND-WDATA FREE-TRAP FROM-ADR FROM-VOC FROM-WORD GET-TCB GET-
TRAP H.  IF-DEBUG IF-TESTING MAP MEMCELLS? NO-EMULATION NOTRAPS
STACK-HOLD STACK.CHECK STACK.MARK START.UNRAVEL.QUIT
STOP.UNRAVEL.QUIT SYSUSER# TIB.DUMP TRAPS UNRAVEL VALID-NAME?
WORD.DUMP
```

Dictionary Address Conversion

```
'>BODY '>NAME >BODY >LINK >NAME BODY> LINK> N>LINK NAME>
```

Dictionary Management

```
' , ALIGN ALLOT C, CRID.  DP DPLIMIT HERE ID.  IMMEDIATE? INITVOCS
MAX-INLINE PAD ROMABLE ROOT W, ['] 
```

DOS Commands from JU:DOSCOMMANDS

```
+DOS >DOS ASSIGN AVAIL CD COPY DATE DELETE DIR DOS DOS0 DOSCOMMAND
DOSSTRING INFO LIST NEWCLI PATH RENAME RUN STATUS
```

Disassembler from JU:DISM

```
ADISM DEF DISM DISM-CYCLES DISM-NAMES DISM-WORD? INIT-DISM RISM
SELECT-DISM-DEFAULTS SEE
```

Defining Words

```
#VOCS : :CREATE ; ARRAY CONSTANT CREATE REVERTVOC USER VARIABLE
VALUE { }
```

Error Reporting

```
.ERR ?ABORT" ?ERROR ABORT ABORT" ERROR MSG QUIT WARNING WARNING"
```

Forth Screen Support

```
BLK BLKERR LIST LOAD R# SCRED
```

Flow of Control

```
+LOOP -DO -LOOP 0BRANCH ?EXIT ?LEAVE ?RETURN ?STAY AGAIN BACK
BEGIN BRANCH CASE CASE_FLAG CONDITION DO DO-LOOP-NEST ELSE END
END-SELECT ENDCASE ENDCOND ENDOF EXIT FORWARD I IF IF-NOT IK J
LEAVE LOOP LOOP-BACK LOOP-FORWARD NOT0BRANCH OF REPEAT RETURN
SELECT THEN TIMES UNTIL UNTIL-NOT WHILE WHILE-NOT
```

FILE I/O

```
$FOPEN +DOS @&CLOSEFILES ACCESS_READ ACCESS_WRITE BLOCK CLOSEFILES
EOF EOL F@, F@,? FCLOSE FCLOSEATBYE FCLOSEVAR FEMIT FERROR
FILEMODE FKEY FDUMP FOPEN FREAD FREAD? FSEEK FSEEK? FTYPE FWRITE
MARKFCLOSE MODE_NEWFILE MODE_OLDFILE NEW OFFSET_BEGINNING
OFFSET_CURRENT OFFSET_END OLD TEMPBUFF TEMPF, TEMPFILE TYPEFILE
UNMARKFCLOSE
```

Floating Point

```
F.  F* FLOAT FNUMBER FNUMBER? FSIN see chapter on Floating Point
```

Forth System

```
BYE COLD INTERPRET QUIT
```

Host Portability

```
'C CELL CELL* CELL+ CELL- CELL/ CELLS CFA->PFA HO.FIND.CFA
HO.FIND.PFA HOST" HO_MAX_INT HO_MIN_INT INLINE+ INLINE> INLINE@
PFA->NFA PICK79 PICK83 REL->USE USE->REL
```

INPUT

```
#TIB >IN ?PAUSE ?TERMINAL BSIN EXPECT FILEWORD HISTORY KEY PARSE
PARSE-WORD QUERY SKIP-WORD? SOURCE SPAN TIB TIB0 TIBEND UNWORD
WORD Y/N
```

I/O General

```
#CHARS ?LETTER ?VISIBLE ASCII BL CONSOLE EMPTYCHAR HICASE
LINELIMIT LOCASE LPLACE LWORD NOCASE OFFSET PLACE
```

Logging to a File from JU:LOGTO

```
$LOGTO LOGEND LOGGED? LOGSTART LOGSTOP LOGTO
```

Logical

```
!BITS +SHIFT -SHIFT AND OR SET-BIT SHIFT WORD-SWAP XOR |
```

Memory Access

```
! +! -> 2! 2@ ? @ ABS! ABS@ ABSC! ABSC@ ABSW! ABSW@ C! C@ CMOVE
CMOVE> D! D@ ERASE FILL MOVE ODD! ODD@ ODDD! ODDD@ ODDW! OFF ON
SPARE TOGGLE W! W@ WMOVE X! X@
```

Memory Allocation

```
+STACK -STACK @&FREEBLOCKS ALLOCBLOCK ALLOCBLOCK? ALLOCSTRUCT
FREEATBYE FREEBLOCK FREEBLOCKS FREEBYTE FREEBYTEA FREECELL FREEMEM
MARKFREEBLOCK MEMF_CHIP MEMF_CLEAR MEMF_FAST MEMF_LARGEST
MEMF_PUBLIC NEXTMEM NEXTMEMA POP PUSH SIZEMEM UNMARKFREEBLOCK
XALLOCBLK XFREEBLK
```

Numeric Conversion

```
# #> #DIGS #DIGITS #S $ (NUMBER) .  .HEX .HX .R <# BASE BINARY
COMMAS CONVERT D.  D.R DECIMAL DIGIT DIGS/, DPL HEX HLD HOLD
N>TEXT NO-COMMAS NUMBER NUMBER? SIGN U.
```

Output

```
+OUT ." .S <FASTEMIT> <FASTKEY> <FLUSHEMIT> >NEWLINE ?MORE ?WRAP
BELL BSOUT CLRCHAR CLS CR CR? EMIT EMIT-TO-COLUMN FAST FLUSHEMIT
MAX-TYPE OUT OUTBUF OUTPUT-CASE SLOW SPACE SPACES TAB TAB-WIDTH
TYPE TYPE-HERE WTYPE
```

Random Numbers from JU:RANDOM

```
CHOOSE RAND-SEED RANDOM WCHOOSE
```

Return Stack

```
.RS .RSTACK 0RP >R DUP>R MYRP R R0 R> R@ RDEPTH RDROP RP! RP@ RP+!
RPICK SET-RP X>R XR> XRDROP
```

Saving and Loading

```
#K #RELOCS #U .IMAGE .RELOCS ?FORGOTTEN ABSRELOCS CLONE FBLK
FILEHEADERS HEADTAIL IFOPTIMIZE INCLUDE INCLUDE? LOAD-FILE
PUSHRELOC REDEF? RELOCSADR SAVE-FORTH SAVE-IMAGE SCR-FILE SCREDING
TURNKEY TURNKEYING?
```

Stack Manipulation

```
-2SORT -DUP -ROT 0SP 2DROP 2DUP 2OVER 2SORT 2SWAP ?DUP ?STACK CSP
DDROP DDUP DEPTH DROP DSWAP DUP MYDSP MYTOS NIP OVER PICK ROT S0
SET-SP SP! SP@ SWAP TUCK XDROP XDUP XPICK
```

Strings

```
" $" $, $- $.  $= $ACCUM $APPEND $CLR $CONCAT $LEN $MOVE $N>S
$SIZE $TYPE $VARIABLE -TRAILING /STRING 0" COMPARE COUNT MAKEUCASE
MATCH? MCASE-SENSITIVE SCAN SKIP SKIP-WORD SKIPWORD TEXT=? UPPER
UPPERC@
```

Timers

```
BENCH BENCH.WITH MEASURE MSEC PROFILE
```

Unused Words Analysis from JU:UNUSED

```
CLEAR.MARKS MARK.UNUSED MARK.USED START.MARKING.WORDS
STOP.MARKING.WORDS UNUSED.WORDS USED.WORDS USED_BIT
```

User Stack

```
>US 'USP> INIT-USP US-DEPTH US-PICK US> US@ USP! USP@ UP0 USP
```

Vectored Execution

```
>IS @EXECUTE COLDEXEC DEFER DEFER-EXECUTE EXECUTE GLOBAL-DEFER IS
ISDEFUSER? WHAT'S
```

Virtual File I/O

```
BUFFERADR CLOSEFVREAD CLOSEFVWRITE F, FFLUSH? LINESFILLV OPENFV
READLINE VIRTBUFFSIZE
```

Vocabulary Management

```
ANEW CONTEXT CURRENT FENCE FIND FORGET FREEZE LATEST MAXVOCS SCAN-
ALL-VOCS SCAN-VOC SCAN-WORDS VALLOT VLATEST>VLINK VLINK>'
VLINK>VLATEST VLIST VOC-LINK VOCABULARY WHEN-SCANNED WHEN-VOC-
SCANNED WORDS WORDS-LIKE
```

Word Sizing

```
B->S S->D W->S
```

Appendix G
Incompatibilities

Between V2.0 and V3.0

Error Handling in IFF and Picture words.

Numerous changes have been made in the way that these words handle errors. IFF.WRITE? has a new stack diagram. These changes are described in detail in a special section on incompatibilities in chapter 21.

Local Variables

The local variable used in V2.0 have been completely replaced. We tried to make the new ones compatible with the old ones but there are some differences. The new locals are kept on the return stack instead of the data stack. The only code that should be affected was code that illegally used internal aspects of the old locals or that illegally relied on the parameters being on the data stack.

Between V1.2 and V2.0

The changes between JForth 1.2 and 2.0 mainly involve the addition of new code. We tried to avoid making any changes that would cause code written using version 1.2 to fail under 2.0. In some cases, however, we felt that the definition of a word in 1.2 was faulty and was worth correcting. Luckily the changes are minor.

>BODY and BODY>

In JForth 1.2, these were defined as NOOP. This was because JForth is subroutine threaded does not have a traditional CFA and PFA. We felt, however, that >BODY was mainly used to get the address of data in a CREATE DOES> word. For JForth, this meant we had to change the definition from NOOP to:

```
: >BODY  ( cfa -- body ) DO-DOES-SIZE + ;
```

Please correct for this in your code if you use >BODY or BODY>.

CONSOLE

This variable contained the address of the file used for I/O by JForth V1.2. Since I/O is a two way street, Input AND Output, we actually need **two** separate variables. This is required to make I/O redirection work properly in Cloned programs. We, therefore, split CONSOLE into CONSOLEIN and CONSOLEOUT. The single word CONSOLE! was added that will set both of these variables. We also added CONSOLE@ for symmetry. CONSOLE@ fetches the value of CONSOLEOUT.

In JForth V1.2:

```
CONSOLE   !
```

In JForth V2.0:

```
CONSOLE!  ( one word )
```

Totally New Floating Point

The old floating point system was not worth keeping after we got a better version from Dave Sirag. We recommend that you switch to this new system. If you absolutely need the old system, take it off of the old disks.

Bugs Fixed

Since bugs were fixed in these words, they will behave differently. You will only have to change your code if you had made a patch for these words. There were other bugs fixed but they should not affect stack diagrams.

```
ROLL  ( was totally messed up, now OK )
U/    ( gave wrong answers sometimes for big numbers )
D!    ( used to swap halves incorrectly, not anymore )
```

Signed AND Unsigned Structure Members

JForth 1.2 treated all structure members as unsigned. The Amiga makes a distinction between signed and unsigned so JForth 2.0 does so as well. The difference is that if you FETCH an 8 or 16 bit signed member, it will be sign extended if it is declared as SHORT or BYTE. Members declared as USHORT or UBYTE will not be sign extended. If you store a -3 in a signed BYTE using ..! then get it back using ..! , you will have -3 on the stack. In an UNSIGNED BYTE you would end up with hex FD.

We have given you an option here. If you do not want automatic sign extension in your program, compile your code with the variable SIGNED-MEMBERS set to FALSE. The default is TRUE.

Appendix G
Incompatibilities

Between V2.0 and V3.0

Error Handling in IFF and Picture words.

Numerous changes have been made in the way that these words handle errors. IFF.WRITE? has a new stack diagram. These changes are described in detail in a special section on incompatibilities in chapter 21.

Local Variables

The local variable used in V2.0 have been completely replaced. We tried to make the new ones compatible with the old ones but there are some differences. The new locals are kept on the return stack instead of the data stack. The only code that should be affected was code that illegally used internal aspects of the old locals or that illegally relied on the parameters being on the data stack.

Between V1.2 and V2.0

The changes between JForth 1.2 and 2.0 mainly involve the addition of new code. We tried to avoid making any changes that would cause code written using version 1.2 to fail under 2.0. In some cases, however, we felt that the definition of a word in 1.2 was faulty and was worth correcting. Luckily the changes are minor.

>BODY and BODY>

In JForth 1.2, these were defined as NOOP. This was because JForth is subroutine threaded does not have a traditional CFA and PFA. We felt, however, that >BODY was mainly used to get the address of data in a CREATE DOES> word. For JForth, this meant we had to change the definition from NOOP to:

```
: >BODY  ( cfa -- body ) DO-DOES-SIZE + ;
```

Please correct for this in your code if you use >BODY or BODY>.

CONSOLE

This variable contained the address of the file used for I/O by JForth V1.2. Since I/O is a two way street, Input AND Output, we actually need **two** separate variables. This is required to make I/O redirection work properly in Cloned programs. We, therefore, split CONSOLE into CONSOLEIN and CONSOLEOUT. The single word CONSOLE! was added that will set both of these variables. We also added CONSOLE@ for symmetry. CONSOLE@ fetches the value of CONSOLEOUT.

In JForth V1.2:

```
CONSOLE   !
```

In JForth V2.0:

```
CONSOLE!  ( one word )
```

Totally New Floating Point

The old floating point system was not worth keeping after we got a better version from Dave Sirag. We recommend that you switch to this new system. If you absolutely need the old system, take it off of the old disks.

Bugs Fixed

Since bugs were fixed in these words, they will behave differently. You will only have to change your code if you had made a patch for these words. There were other bugs fixed but they should not affect stack diagrams.

```
ROLL  ( was totally messed up, now OK )
U/    ( gave wrong answers sometimes for big numbers )
D!    ( used to swap halves incorrectly, not anymore )
```

Signed AND Unsigned Structure Members

JForth 1.2 treated all structure members as unsigned. The Amiga makes a distinction between signed and unsigned so JForth 2.0 does so as well. The difference is that if you FETCH an 8 or 16 bit signed member, it will be sign extended if it is declared as SHORT or BYTE. Members declared as USHORT or UBYTE will not be sign extended. If you store a -3 in a signed BYTE using ..! then get it back using ..! , you will have -3 on the stack. In an UNSIGNED BYTE you would end up with hex FD.

We have given you an option here. If you do not want automatic sign extension in your program, compile your code with the variable SIGNED-MEMBERS set to FALSE. The default is TRUE.

Appendix H
Products Written in JForth

This chapter describes commercial products that have been written in JForth. This is a partial list and contains only the ones we know about. If you have any JForth programs that you are selling, send us information and hopefully a sample copy, and we will, if possible, include a description for free in our next manual. If you read about something here that you like, please help your fellow JForth programmers by buying it.

B.A.D. from MV Micros

This "best selling utility" optimizes disk access time up to 500% by analyzing and restructuring any Amiga DOS disk. B.A.D. can reduce "disk gronking", speedup window opening under Workbench, and speed up DIR and other disk operations. B.A.D. also includes a disk structure test to detect errors. Works on both floppy and hard disks. B.A.D. is available from many mail order outlets. It is distributed by Centaur Software, Inc.

My Diary from MV Micros

My Diary is a full featured Personal Information Manager (PIM). My Diary provides a pop-up notepad for managing names, phone numbers, addresses, appointments, "to do" lists, due dates, diaries, birthdays, etc. Notes are date stamped and can be attached to the calendar. Notes can be searched for keywords. Timers can be used to provide *reminders*. Information can be imported from and exported to hand held electronic organizers like the Sharp Wizard. ARexx support and Auto Dialer also provided.

HMSL, the Hierarchical Music Specification Language

HMSL is a programming language for experimental music composition and performance. It is an extension to JForth that adds MIDI capability, local sound support, and an extensive set of object-oriented ODE classes for organizing, manipulating and playing musical data. HMSL also includes a graphics toolbox for building interactive screens, a score entry dialect, and a simple sequencer. HMSL was developed by Phil Burk, Larry Polansky, and David Rosenboom. For information contact:

```
Frog Peak Music
P.O. Box 151051
San Rafael, CA
94915-1051
(415) 461-1442
```

Copyist Companion by Nick Didkovsky

Copyist Companion converts Deluxe Music Construction Set (DMCS) score files to Dr. T's The Copyist files. All the features of the DMCS score are meticulously converted, including time and key signature changes, clefs, ties, beaming, stem direction, slurs, dots, chords, triplets, quintuplets, accidentals, dynamics, etc. The result is a beautifully laid out Copyist score, ready for professional printing. Copyist Companion was written in JForth (of course) and runs on any Amiga. The great quantity of reverse engineering required to parse the DMCS file format made JForth and ODE the only programming environment I would have considered using for this project. You can get more info about Copyist Companion from:

```
Nick Didkovsky
171 East 99th Street #20
New York, NY 10029
(212) 369-1733
email didkovsk@dorsai.com
```

Or contact Dr. T's Music Software, who are distributing Copyist Companion.

Nick is also developing a DMCS to MIDIfile conversion utility. Call or write for details on its progress.

Doctor Nerve with Nick Didkovsky

Doctor Nerve's latest two CD's are "Beta 14 ok" and "Did Sprinting Die?". Both projects include compositions generated by HMSL (thus the JForth connection). These compositions are performed by the band on "Beta" and directly by Amiga/HMSL on "Sprinting". Both CD's also include non-HMSL-composed material. See if you can tell the difference!

Doctor Nerve LP's and CD's can be ordered from:

```
Wayside Music
PO Box 6517
Wheaton, MD 20906.
```

Ask for their catalog. Or contact Nick Didkovsky as above.

IntuiEZ by Curtis Stanton

IntuiEZ is a shareware program that allows the user to interactively design screens and windows with gadgets using the mouse. You can then output JForth compatible source code to a file. IntuEZ also has the ability to capture the Intuition structure from other applications screens and window for study or modification. This can create gadgets with the version 2.0 style 3D gadgets but it.works with Amiga DOS 1.3. At press time it was not known how this program would be distributed so call Delta Research for the latest information, or look for it on a BBS.

XL by Martin Kees

XL is a cel animation program by the author of JForth's ANIM support routines. It uses an *onion-skin* display that lets you see several cels at once. It also features a very powerful bezier curve editor. A

demonstration version of XL can be found on Fred Fish disk #516 which, by the way, is entirely filled with programs written in JForth. On the same disk check out Enigma, a beautiful puzzle that models a programmable machine.

JGoodies_1 from various.

Check out Fred Fish disk #239. It is also all JForth code. Includes a fast Mandelbrot generator, audio tools, FFT code, HeadClean, and other goodies.

Index

$FOPEN, 8-4
'O, 10-22
..! , definition,18-10
..@ , definition,18-10
0FOPEN, 8-4
:CLASS, 10-16
:M, 10-16
<SUPER, 10-16
>ABS with libraries, 18-2
>ABS, 18-13
>ABS, ORG, 14-2
>REL, ORG, 14-2
?INSTANTIATE:, 10-22
?INSTANTIATE:, OB.OBJLIST, 10-15
?NEW:, OB.BARRAY, 10-9
?NEW:, OB.ELMNTS, 10-11
[], Tutorial, 10-5
[COMPILE], 6-4
[FORGET], 11-6

A

AbortIO(), 24-6
Absolute value, ABS, GL-30
ADD:, OB.ELMNTS, 10-11
Address conversion for libraries, 18-2
Address conversion in assembler, 14-8
Address conversion, >ABS , GL-25
Address conversion, structures, 18-9
Address relocation, 12-5
Address, absolute, 14-2
Address, automatic call conversion, 18-5, 6
Address, conversion, 18-13
Address, even, ALIGN, GL-31
Addresses, relative vs absolute, 2-6
Alert, RUDE, AP-6
Algebraic expressions, 4-6
ALL.METHODS, 10-21
Allocate dictionary, ALLOT, GL-32
Allocate/instantiate objects 10-15
Allocate, structure, 18-8
ALLOCBLOCK, 11-1
AllocMem(), ALLOCBLOCK, GL-31
ALLOT, USE.DICT:, 10-10
AmigaDOS 2.0, 24-8
ANEW, 11-5
ANEW, tutorial, 5-2
ANIM.xxx, definitions, 22-6
Animation, 22-1
Animation, moving brush, 21-4
Animbrush, tutorial, 22-4
Application Specific Library, 24-9
Applications, stand-alone, 7-1
APTR, 18-9
Area, 24-4
ARexx Arguments, 23-3
ARexx Support, 23-1
ARexx Variables, 23-8
ARGS, library calls, 18-1
Arithmetic, 4-4
Array object, 10-3
Array of structures, 18-8
Array, bytewide, CARRAY, GL-41
Array, OB.BARRAY 10-9
ARRAYOF structures, 18-11
Arrays in structures, 18-9
ASCII characters, 4-7
ASL, 24-9
Assembler module, 16-2
Assembler, 68000, 14-1
Assembler, clone restrictions, 7-7
Assembler, macros, 14-5
Assembler, Motorola style, 14-6
Assembler, RPN, 14-2
ASSIGN, 3-1
AT:, OB.BARRAY, 10-9

B

Backspace char, BSOUT, GL-38
BACKWARD:, OB.ELMNTS, 10-11
BASE in #, GL-3
BASE, numeric, 5-10
BBS, Forth support, AP-2
BEGIN UNTIL, 5-7
BeginIO(), 24-6
Benchmark, 2-4
Benchmarks, 3-2
Binary files, 8-2
Binding, ODE, 10-5
Bitmap allocation, 24-1
BLOCK files, 15-1
BLOCK, convert to text file, 15-4
BLOCK, files, 3-3
BLOCK2TEXT, file conversion, 3-3
Boolean operators, 5-5
BOTH, 12-5
Breakpoint, debug, 13-4
Breakpoint, setting, 13-6
Brush, dump, AP-6
Buffered file I/O, 8-7

C

CALL, libraries, 18-4
CALLED, 12-5
Calling a library, example, 18-1
CASE, 5-7
CELL, concept, AP-10
CFA, , 12-5
CFA, 12-4
CFA, get with ', GL-6
Change directory, CD, GL-41
CHIP memory, ALLOCBLOCK, GL-31
CHOP:, OB.ELMNTS, 10-11
Chunks in IFF files, 21-10
Class, definition, 10-2
Class, example, 10-19
Classes, defining new, 10-16
CLEAR:, OB.BARRAY, 10-9
CLEAR:, OB.INT, 10-9
Clipping, PIC, 21-6
Clone and modules, 16-5
Clone, 2-2
Clone, 7-1
Clone, debugging, 13-5
Cloning ODE programs, 10-23
Close file, 8-6
Code Field, 12-4
Color of input text, 17-1
Color, index, 19-2
Color, text, foreground, 24-8
Compare numbers, 5-5
Compile CFA reference (CFA,), GL-9
Compile from Textra, 23-7
COMPILE, 6-7
Compile, INCLUDE, 2-2-5
Compiler, 12-5
Compiler, STATE, GL-95
Compiling a program, 5-2
Compiling IMMEDIATE words, 6-7
Compress bitmap, 21-16
Conditional compilation .IF, GL-17
Conditionals, 5-6
Conditionals, style, AP-10
CONSOLE, cloned, 7-3
CONTEXT, 17-3
Control-C, cloned, 7-3
Control-D, 13-5
Convert .h files, 18-, 18-14
CREATE DOES>, 6-6
CREATE example, 6-1
CREATE, data, GL-25
CreatePort(), 24-6
CreateStdIO(), 24-6
CURRENT:, OB.ELMNTS, 10-11
Cursor control, 24-7
Cursor keys, history, 17-1

D

Data registers, 14-1
Data stack, 12-2
DATA.ADDR:, OB.BARRAY, 10-9
DEBUG, UNRAVEL calls, GL-101
Debugger, tutorial, 13-3
Debugging, tools, 13-1
DEBUG{, 13-6
DEF, 14-11
DEFER, 11-3
Defer, cloned, 7-7
Defer, in modules, 16-4
DEINSTANTIATE, 10-22
DEINSTANTIATE:, OB.OBJLIST, 10-15
Delay(), 18-1
DELETE:, OB.LIST, 10-14
Demos, list of, 3-2
Dial, AP-5
Dictionary layout, 12-2
Dictionary pointer, HERE, GL-66
Dictionary size, #K, GL-4
Dictionary, WORDS, 2-3
Dimension, OB.ELMNTS, 10-11
DIMENSION:, OB.ELMNTS, 10-11
Disassembler, 14-9
Divide using / , GL-18
Division, floored, 2-7
DO LOOP, 5-7
DO.RANGE:, OB.BARRAY, 10-9
DO:, OB.ELMNTS, 10-12
Documentation generator, AP-6
DOLINES, 8-8
DOS commands, 2-4
DOS Library, 18-1
DOS, execute command, $DOS, GL-5
Double Buffer, 3-5
Double buffering, 21 5
Double Buffers, in ANIMs, 22-2
Double precision, CALL, 18-5, 6
DSP, 18-9
DUMP.SOURCE:, OB.ELMNTS, 10-12

E

ED.AT:, OB.ELMNTS, 10-12
ED.TO:, OB.ELMNTS, 10-12
ED2I:, OB.ELMNTS, 10-12
Editing text file, 5-1
EDITOR, for BLOCKs, 15-2
Element, OB.ELMNTS, 10-11
EMIT, cloned, 7-3
EMIT, defer, 11-4

EMIT, LOGTO, 11-8
EMPTY:, OB.ELMNTS, 10-12
ERR%, 11-9
Error checking in ODE, 10-21
Error codes, 11-9
Error codes, 23-3
Error handling, 11-9
Error report, .ERR, GL-17
Error reporting in ODE, 10-23
Error, Index out of range, 10-4
Errors, IFF, 21-5
Escape characters, ANSI, 24-7
EV.xxx, definitions, 19-6
Event input, 19-6
Events, menu example, 20-2
Exec support, 24-5
EXEC:, OB.ARRAY, 10-10
EXECUTE, 6-3
EXECUTE, clone restrictions, 7-7
Execution Array, 10-11
Exit JForth, BYE, GL-38
EXPECT, History, 17-2
EXTEND:, OB.BARRAY, 10-9

F

F, , 8-7
Fadein, 21-4
Fadein, PIC.FADEIN, 21-9
FCLOSE, 8-6
FIG standard, AP-12
FIG, Forth Interest Group, AP-2
File creation, 8-1
File names, 8-6
File requester, ASL, 24-9
File, $FOPEN, GL-5
File, buffered, 8-7
File, example, AP-9
File, I/O, 8-1
FILE?, 18-8
Files, closed if TRACKING ON, 7-3
Files, JForth disks, 3-1
FILEWORD, 8-4
FILL.DIM:, OB.ELMNTS, 10-12
FILL:, OB.BARRAY, 10-9
FIND, cloned, 7-6
FIRST:, OB.ELMNTS, 10-12
FIRST:, OB.ELMNTS, 10-12
Flip pictures, 21-9
Floating Point, 9-1
Font Requester, 24-9
Font, 19-5
Font, garbled?, 13-3
Fonts, 24-3
FOPEN, 8-4
Forbid(), 24-7
FORGET using ANEW, GL-32
FORGET, 11-5
FORGET, tutorial, 5-2
Forth Interest Group, AP-2
Free, various, 24-2
FREE:, example, 10-5
FREE:, OB.BARRAY, 10-9
FREEALL:, OB.OBJLIST, 10-15
FREEBLOCK, 11-1
Function keys, 17-2

G

Gadgets, 24-3
GET.NAME:, OBJECT^-8
GET:, OB.ELMNTS, 10-12
GET:, OB.INT, 10-9
GOTO.ERROR, 11-9
GOTO:, OB.ELMNTS, 10-13
GR.xxx, definitions, 19-3
Graphics, misc., 24-1
Graphics, toolkit, 19-1

H

H2J, 18-14
Hash dictionary, 2-5
Header structure, 12-2
Hex entry using $, GL-4
Hide, SMUDGE, 12-3
HISTORY, 17-1
HMSL, AP-1
HMSL, plug, AP-20
Hunks, SHOWHUNKS, 17-5

I

I2ADDR:, OB.ELMNTS, 10-13
I2ED:, OB.ELMNTS, 10-13
IDCMP events, 19-6
IF, style, AP-10
IF, tutorial, 5-6
IF.FORGOTTEN example, AP-9
IF.FORGOTTEN, 11-6
IF.FORGOTTEN, and FREE:, 10-5
IF.FORGOTTEN, with DEFER, 11-4
IFF file support, 21-1
IFF Parser, 21-10
IFF, ANIM formats, 22-1
IFF, dump file, AP-6
IFF, support files, 3-5
IFF.xxx definitions, 21-14
ILBM parsing, 21-11
ILBM, display, 21-1
IMMEDIATE words, compiling, 6-7
IMMEDIATE, 6-6

INCLUDE from Textra, 23-7
Include files, .h and .j, 18-7
Include files, module, 16-3
INCLUDE, 2-5
INCLUDE, echo, 13-1
INCLUDE, tutorial, 5-2
INCLUDE?, tutorial, 5-2
Incompatibilities, 21-16
Index of DO LOOP, 5-8
INDEXOF:, example, 10-4
INDEXOF:, OB.BARRAY, 10-9
Inheritance, 10-1
Inheritance, definition, 10-3
Inheritance, how it works, 10-23
INHERITANCE.OF, 10-21
INIT:, 10-19
Initialization, AUTO.INIT, GL-34
Initialization, compile time, AP-9
Initialize Forth,COLD, GL-44
INLINE data, AP-11
INLINE, 12-5
Input a string, EXPECT, GL-58
Input, TIB, >IN , GL-25
INSERT:, OB.ELMNTS, 10-13
Install on hard disk, 1-2
Instance objects, 10-20
Instance Variable definition, 10-3
Instance variables, 10-17
Instantiate objects in list, 10-15
Instantiation, dynamic, 10-22
INTERPRET, cloned, 7-6
Interrupt, assembly tips, 14-9
Intuition, misc24-1
IV.LONG, 10-16
IV.SHORT, 10-16
IV=>, 10-17

J

JShow, 21-13
Jump table, 10-10

K

Kernel, 12-4
KEY, cloned, 7-3

L

LAST:, OB.ELMNTS, 10-13
Late binding, ODE, 10-5
Libraries, adding new, 18-6
Libraries, Graphics, 19-1
Libraries, Intuition, 19-1
Libraries, tutorial, 18-1
Library, base pointers, 18-4
Library, new, GL-23
LIMIT:, OB.BARRAY, 10-9
Link Field, 12-3
Link field, 6-4
Linked lists, exec, 24-6
List object, OB.LIST, 10-14
List, OB.OBJLIST, 10-14
LITERAL, 6-7
Loader, 12-5
Local Variables and ODE, 10-6
Local Variables, 11-6
Local variables V2 vs V3, AP-17,18
Logical operators, 5-5
LOGTO, 11-8
Loop using BEGIN, GL-35
Loops, tutorial, 5-7

M

MANY:, OB.ELMNTS, 10-13
MANYLEFT:, OB.ELMNTS, 10-13
MARKFCLOSE, 8-6
MAX-INLINE, 7-2
MAX-INLINE, 12-4, 5
MAX.ELEMENTS:, OB.ELMNTS, 10-13
MAXimum, 4-5
Memory allocation problems, 13-2
Memory allocation tracking, 12-5
Memory Allocation, 11-1
Memory allocation, ALLOCBLOCK, GL-31
Memory fragmentation, 1-3
Memory Map, 12-1
Memory, allocation example, AP-9
Memory, CHIP, ODE, 10-23
Memory, FREE:, 10-9
Memory, FREEBLOCK, GL-63
Memory, freed if TRACKING ON, 7-3
Menu defaults, 20-6
Menu, example, 20-1
Menu, EZMENU System, 20-1
Menu, Intuition calls, 20-6
Message Port, ARexx, 23-2
Message, wait, 19-7
METHOD, declaration, 10-17
Method, definition, 10-2
METHODS.OF, 10-21
Module, ASSEM, 14-6
Module, INCLUDES, 19-1
Modules, 16-1
Modules, custom, 16-3
Modules, structures, 18-7
Moore, Charles, 2-1
Mouse input, 19-6
Multitasking, 12-1

N

Name Field, 12-3
NAME:, OBJECT^-8
Naming conventions, AP-8
NEW:, OB.BARRAY, 10-9
NEW:, OB.ELMNTS, 10-13
NEW:, OB.LIST, 10-14
NewWindow, 19-1
NEXT:, OB.ELMNTS, 10-13
NEXTWRAP:, OB.ELMNTS, 10-13
NFA, 12-3
NOT, standards, 2-6
Number compilation, LITERAL, GL-71
Numeric BASE, GL-35
Numeric conversion, GL-3

O

OB.ARRAY, tutorial, 10-3
OB.BARRAY, 10-9
OB.ELMNTS, 10-11
OB.REPORT.ERROR, 10-23
OBJECT class, 10-8
Object, definition, 10-2
ODE, 10-1
ODE, cloning, 7-6
ODE, loading, 10-3
Open file, FOPEN, 8-4
OPENFV, 8-7
OpenLibrary(), 18-6
Optimize, INLINE, GL-69
Optimizer, global, 17-6
ORG register, 14-2
Overflow, */, 4-6

P

PAD, used by ", GL-3
Parameter Field, 12-4
Performance Analyser, 17-7
PFA, 12-4
PFA, >BODY , GL-25
PIC.xxx, definitions, 21-6
PICK, tutorial, 4-3
Pointer to function, 6-3
Polygons, 24-4
Port, FindPort(), 24-5
Portability, AP-10
Print number, . , GL-16
PRINT, application, AP-6
PRINT.DIM:, OB.ELMNTS, 10-13
PRINT:, OB.ELMNTS, 10-13
Printing output, 11-8
PROFILE, 17-7
PUSH, 11-2
PUT.NAME:, OBJECT^-8
PUT:, OB.ELMNTS, 10-13
PUT:, OB.INT, 10-9

Q

Quit JForth, BYE, GL-38
QUIT, cloned, 7-6
Quitting JForth, BYE, 1-2

R

Random, CHOOSE, GL-42
Random, example, 19-2
Range check, objects, 10-4
RANGE:, OB.BARRAY, 10-10
Rastport allocation, 24-1
RastPort, 19-3
Read file, FREAD, GL-63
READLINE, 8-7
Reboot, 1-3
ReCompile JForth, 12-6
Register, dump, 13-7
Registers 68000, utilization, 14-1
Relocation Table, 12-5
Remainder, /MOD, 4-5
REMOVE:, OB.ELMNTS, 10-13
Requester, AUTO.REQUEST, GL-35
RESET:, OB.ELMNTS, 10-13
Return stack, 6-5
Return stack, >R , GL-26
Return stack, debugging, 13-7
Reverse Polish Notation, 4-4
RexxView, 23-10
RPN, 4-4
RUN.FASTER, 10-21
RUN.SAFER, 10-21
RX.ADD.COMMAND, tutorial, 23-4
RX.EXEC.LINE, 23-5
RX.INIT, tutorial, 23-2
RX.xxx, ARexx glossary, 23-5

S

S!, 18-8
SAVE-FORTH and Modules, 16-1
SAVE-FORTH, 2-5
SAVE-FORTH, 6-2
SAVE-FORTH, and #K, GL-4
SCRED, for BLOCKS, 15-3
Screen tools, 24-4
Screen, SIFF-SCREEN, 22-6
Screens, with $PIC.LOAD? 6
Seek file, FSEEK, GL-65
SELF, in methods, 10-17
Serial, Dialer, AP-5
SET.MANY:, OB.ELMNTS, 10-13

SET.WIDTH:, OB.BARRAY, 10-10
Shell, startup, 1-3
SHIFT, 4-6
SHOWHUNKS, 17-5
Sign extension, B->S , GL-35
Sign, words in structures, 18-11
Signals, alloc, 24-5
Size Field, 12-4
Size, dictionary, #K, GL-4
Size, dictionary, cloned, 7-4
SIZE:, OB.ELMNTS, 10-14
Smalltalk, 10-2
SMEAR:, OB.ELMNTS, 10-14
SMUDGE bit, 12-3
SMUDGE in colon, GL-23
Sort, -2SORT, 5-8
Sort, Batcher, BSORT, GL-38
Sortmerge, AP-7
Speech, SayNumber.f, AP-7
SPLIT:, OB.ELMNTS, 10-14
Stack diagram, 4-2
Stack location, 12-2
Stack size, 2-6
Stack, dynamic, 11-2
Stack, tutorial, 4-1
Standards, Forth, AP-12
STATE of the compiler, 6-6
STRETCH:, OB.ELMNTS, 10-14
String append, 6-2
String compare, $-, GL-5
String compare, MATCH?, GL-75
String comparison, 6-1
String handling, 6-1
String input, EXPECT, GL-58
String print, ." , GL-16
String, $APPEND, GL-5
String, ", GL-3
String, compile using $, , GL-5
String, NUL terminated, 0" , GL-19
String, NUL terminated >DOS, GL-25
Strings, 5-9
Structure unions, 18-12
Structure, access, 18-8
Structure, assembler access, 14-8
Structure, defining, 18-9
Structure, dump, 18-9
Structure, passing, 19-1
Structures, 18-7
Structures, array of, 18-8
Structures, nested, 18-8
Submenu, 20-5
Subroutine threading, 2-6
SUPER, in methods, 10-18
Superclass, <SUPER, 10-16
Superclass, definition, 10-2
Switch(), CASE, GL-41

T

TAB-WIDTH, 17-3
Tag lists, 24-9
Terminal, AP-7
Text editor, ARexx, 23-7
Text I/O, 5-9
Text justification, 19-5
Text, color, foreground, 24-8
Textra, ARexx, 23-7
Textra, install, 2-3
Time, BENCH, 2-4
Time, example, 18-2
TO:, OB.BARRAY, 10-10
Top of Stack, cache, 14-2
Trace, UNRAVEL, 13-1
Trailing space removal, GL-16
Transportability, AP-10
Turnkey, 7-1

U

UNMARKFCLOSE, 8-6
UNRAVEL, 13-1
Update, AP-7
Upper case, MAKEUCASE, GL-74
USE.DICT:, OB.BARRAY, 10-10
USED.WORDS, 11-9
USER and modules, 16-4
USER Variables, 12-1
USER variables, cloned, 7-8
Utilities, files, 3-6

V

VARIABLE, from assembler, 14-9
Variables, 5-3
Variables, ARexx, 23-8
Vectored execution, 11-3
VERIFY-LIBS, 18-5
VERIFY-LIBS, cloned, 7-8
Version, libraries, 18-3
Vocabularies, 17-3
VOCABULARY and modules, 16-4

W

WHERE:, OB.ELMNTS, 10-14
WIDTH:, OB.BARRAY, 10-10
Window of JForth, GL-107
Window, example, 14-9
Window, example, 18-13
Window, menus, 20-3
Window, NewWIndow, 18-8
Window, open, 19-1

Wipe picture, 21-4
Wipe, PIC.WIPE, 21-9
WORD, cloned, 7-6
}STUFF:, OB.BARRAY, 10-10
}STUFF:, tutorial, 10-5